Sistemas Informáticos
Curso 2002-03

PROYECTO ClueX
Gestor Inteligente de Cálculo Distribuído

Luis Domínguez Roldán
Daniel Navas-Parejo Alonso
Juan Antonio recio García

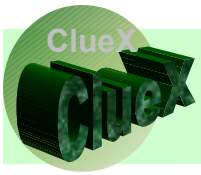
Dirigido por:
Prof. Luis Javier García Villalba
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

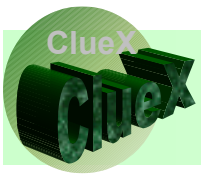
Los abajo firmantes: Daniel Navas-Parejo Alonso, Juan Antonio Recio García y Luis Domínguez Roldán, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, y, mencionando expresamente a sus autores, tanto la presente memoria, como el código, la documentación, y/o el prototipo desarrollado.

Luis Domínguez Roldán Daniel Navas-Parejo Alonso Juan Antonio Recio García

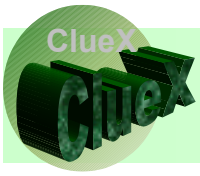
Indice de contenidos



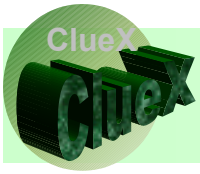
Indice de contenidos



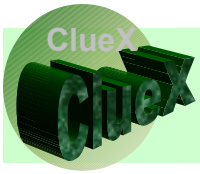
Indice de contenidos.

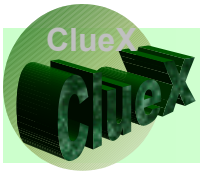


Indice de contenidos.



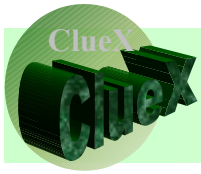
Indice de contenidos.





MÓDULO I

PROPUESTA



1.1.- Propuesta inicial.

El objetivo del proyecto es la implementación de una arquitectura distribuida, y de algoritmos de Inteligencia Artificial que realicen las decisiones de Gestión de procesos en ella.

La **arquitectura** está formada por:

- Un planificador, que ejecuta el programa gestor de procesos “AIEngine”.
- Una serie de máquinas-cliente, solicitantes de tareas a ejecutar.
- Una serie de estaciones de cálculo, encargadas de ejecutar las órdenes.

El **programa gestor** se basa en los siguientes aspectos:

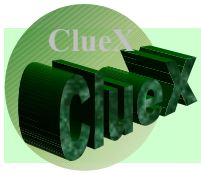
- Uso de Inteligencia Artificial para realizar las decisiones, implementando algoritmos de planificación basados en diversas políticas, y realización de una Base de Datos. Concretamente:
  - Round Robin.
  - Razonamiento basado en casos.
  - Balanceo de Carga.
  - Red Neuronal.

Por su parte, las **estaciones de cálculo** son capaces de realizar las siguientes funciones:

- Recoger estadísticas
- Avisar de los distintos eventos de ejecución (inicio, final, errores...)

Se ha añadido al proyecto las siguientes **funcionalidades**:

- Cambio de algoritmo de planificación “en caliente”.
- Realizar una gestión no centralizada en un servidor mediante una GUI
- Encriptación de las comunicaciones.



1.1.- Initial proposal.

The aim of the proyect is the implementation of a distributed system, and AI algorithms able to make the process management in it.

The **architecture** is composed by:

- A scheduler machine, running the process manager, called “AIEngine”.
- A group of clients, demanding the execution of any process.
- A group of “calc stations”, executing the commands.

The **Process Manager** is based in:

- Use of AI to make decisions, with process management algorithms based on different policies and a Database Manager. More specifically:
  - Round Robin.
  - Case Based reasoning.
  - Qualified Load Balancing.
  - Neural Network

Calc Stations are able to:

- Harvest statistics.
- Warn about different execution events (beginnings, endings, errors,…)

Some **other functionality** has been added to the proyect:

- Support of “hot” policy change.
- Uncentralized management of the system using a GUI
- Safe (encrypted) comunications.

1.2.1.- Propuesta detallada: Presentación.

1.2.1.1.- Introducción.

El proyecto **ClueX** (Cluster Experimental) consiste en un conjunto de máquinas de bajo costo que intenta hacer el papel de Application Server para un grupo de clientes dentro de una LAN. Se puede ver como una fusión entre conceptos de **programación de sistemas, de aplicaciones, inteligencia artificial y arquitectura de red.**

Lo presentamos como un planificador inteligente de procesos sobre una arquitectura distribuída de bajo costo, **balanceando carga de CPU/MEM** en sus nodos, de tal forma que está compuesto de:

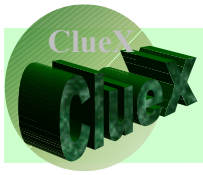
- **Planificador** configurable.
- **Base de Datos** del planificador: donde se guardará la Base de Conocimiento del sistema, con parámetros objetivos como velocidad de CPU, y subjetivos, como puede ser si el día es festivo, o el tiempo que va a hacer, que influyen indirectamente en la cantidad de carga de procesos.
- **Agentes recolectores de estadísticas** y datos de interés para el planificador.
- **Agentes Software** encargados de optimizar los SO de los nodos del cluster según las necesidades de los procesos lanzados.
- **Software de interconexión de red**.
- **Interfaz** (quizá una GUI web) para la parametrización del planificador y el lanzamiento de procesos por parte de las máquinas cliente.

En el proyecto incluiremos un conjunto de "procesos tipo" con los que realizaremos una batería de pruebas de rendimiento y comportamiento.

1.2.1.2.-Objetivo y alcance del Proyecto ClueX.

ClueX está pensado para obtener rendimiento de máquinas anticuadas, poniéndolas a trabajar en conjunto. El **objetivo** es pues que se obtenga mejor rendimiento en N máquinas (que llamaremos m) de ClueX que en una sóla máquina independiente de mayor potencia (la llamaremos M), de tal manera que N venga dado por:

$$N*(p_m + Ge_m) + P_i = P_M$$



Donde:

P_M = precio_de_mercado_de_M
p_m = precio_de_mercado_de_m
Ge_m = gasto_eléctrico_de_m
P_i = precio interconexión

Queda fuera del alcance del proyecto la paralelización de aplicaciones. Por lo tanto, este objetivo de escalabilidad se intentará conseguir en condiciones de saturación de capacidad de proceso, es decir, no pondremos como prueba un conjunto de procesos pequeño que se puedan ejecutar a la vez en una máquina sin interferencia de rendimiento entre ellos, sino que más bien, queremos una carga de procesos de un tamaño tal que ejecutándose a la vez, tarden más que ejecutandose por separado en esa única máquina.

El cluster ClueX está pensado para que las máquinas clientes sean a su vez **"servidores de terminales",** para aprovechar más la potencia de cálculo del cluster.

1.2.1.3.- Relación de ClueX con las materias de la carrera.

El proyecto ClueX comprende un conjunto de conocimientos que están relacionados muy estrechamente con las materias de **"Sistemas Operativos"**, **"Inteligencia Artficial e Ingeniería del Conocimiento"**, **"Bases de Datos y Sistemas de la Información"**, **"Redes"**, **"Metodología y Técnicas de Programación"** y el compendio de materias de **Programación.** También, ya que se va a proceder al Benchmarking del cluster, se relaciona con **"Evaluación del Rendimiento de Configuraciones".**

Dentro de estas materias, podemos especificar un poco más:

- Algoritmos de planificación: SO, IAIC, MTP.

- Benchmarking: ERC, SO, REDES, MTP (Algoritmos de ejemplo).

- Base de Datos del planificador: BDSI, IAIC (Tratamiento del aprendizaje).

- Consola de Gestión del planificador: Programación.

- Comunicación - Sockets - Arquitectura: REDES.

- Agentes Software: IAIC.

1.2.1.4.- Posibles Ampliaciones.

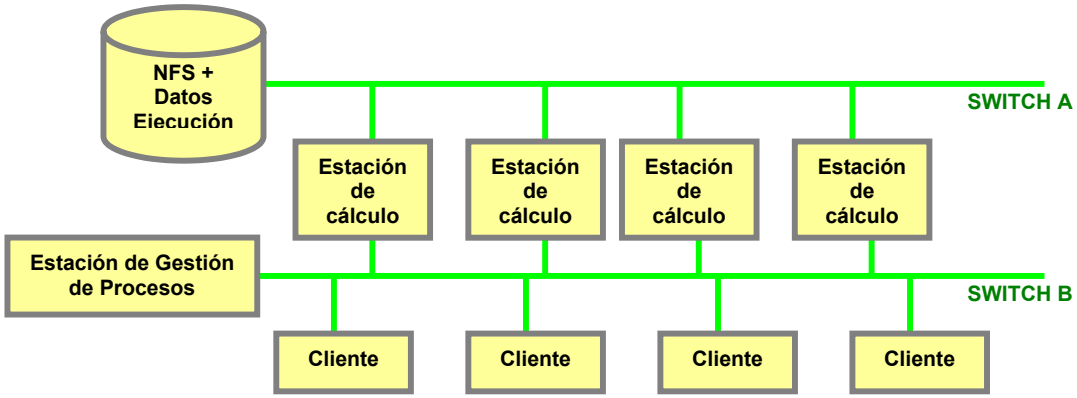
- **Funcionalidades de los Agentes Software**: Realizar un Data Mining más exhaustivo a partir de las relaciones *"proceso-proceso"* (cada vez que arranco tal proceso casi siempre se arranca este otro) y *"usuario-proceso"* (el usuario X casi siempre usa el mismo tipo de aplicaciones) (**IAIC - BDSI**)

- Paralelismo a nivel de datos (*High Thoughput Computing*): Para Software específcamente diseñado para esta arquitectura.

- **Paralelismo a nivel de proceso o a nivel de thread** (HPC).

- Descentralización de la planificación.

1.2.2.- Propuesta detallada: Arquitectura.



1.2.2.1.- Estación de Gestión de procesos.

- Hace el papel de **iniciador** de los procesos
- Casi todo el código fuente del gestor de procesos se encuentra aquí.
- Consta de varios componentes:

  - **Agente recolector de estadísticas**: Escucha en la red los mensajes que lleguen de las estaciones de cálculo, que contienen información útil para las tareas de planificación.

  - **Planificador:** Usa los mensajes procedentes de las estaciones de cálculo y su propia base de datos de procesos para inferir cuál será la mejor elección a la hora de lanzar el próximo proceso. Se basará en parámetros como carga de CPU, memoria disponible, tamaño del proceso, historia del proceso, I/O involucrada en el proceso, etc… que guarda en una Base de Datos en la misma máquina. Se puede definir como un Agente Software que monitoriza los recursos del cluster y reparte los procesos según convenga en cada momento. También se encarga de mover procesos entre estaciones en el caso de que un error ocurra, o falten recursos.

  - **Consola de gestión:** Aplicación para la gestión del planificador, ajuste de parámetros, etc… También incluye el interfaz de comunicación con los clientes.

  - **Alternativas posibles:** la consola de gestión puede estar ubicada en otra máquina, conectándose a la consola de gestión por una red separada, descargando así al planificador del consumo de CPU/MEM que implica una GUI.

1.2.2.2.- Almacenamiento compartido.

- El resultado de los procesos debe guardarse en un lugar común, pues no tiene sentido ir a buscarlo a cada una de las estaciones de cálculo.

- Los binarios de los procesos deben estar instalados todos aquí.

- **Lo ideal**: tener una máquina enteramente ocupada con esta tarea.

- **Alternativa**: Fusionar esta funcionalidad en la máquina que hace de estación de gestión.

- **Opciones de rendimiento**: Lo ideal sería tener un sistema RAID 5, que nos da alta disponibilidad y alto rendimiento, pero si pensamos en usar máquinas PC debemos pensar en que es más práctico usar RAID 0, que da alto rendimiento y no alta disponibilidad, ya que nuestro objetivo es mejorar el tiempo de ejecución del sistema multiprocesador, pues para usar RAID 5 deberíamos tener al menos 3 discos. EL no tener alta disponibilidad no afecta demasiado ya que éste no es un sistema en producción.

- **Alternativas de arquitectura**: RAID Hardware (lo ideal) o Software.

1.2.2.3.- Estaciones de cálculo.

- Aquí se llevan a cabo las tareas reales de proceso.

- **Objetivo**: Escalar en una relación razonable el tiempo de ejecución de un conjunto de procesos que por sí solo satura la capacidad de proceso de una sola máquina.

- **Alternativas**: Computación a nivel de aplicación (opción básica), de proceso, o a nivel de threads (opción ideal). Queda fuera del alcance de este proyecto la computación a nivel instrucción máquina.

- **Semejanzas en la arquitectura:** Pensamos que esta arquitectura debe intentar imitar en lo posible un cluster en modo Scalable y/o un software de Application Server.

- En cada estación de cálculo instalaremos un **agente** de tamaño reducido que se limite a **recoger información** de la máquina para pasarla al planificador de la Estación de Gestión. Un protocolo interesante para esta comunicación sería SNMP (Simple Network Management Protocol), aunque no lo implementaremos, quizá se use la idea general, o quizá usemos freeware.

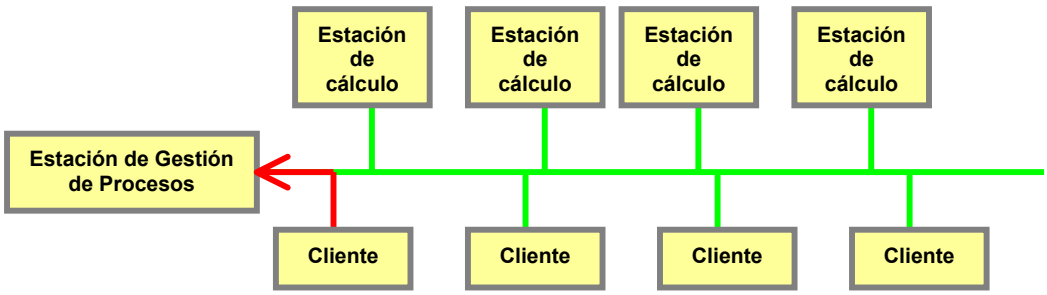
1.2.2.4.- Clientes.

- Simplemente son las máquinas que van a aprovechar la capacidad de cálculo del cluster, ya que son las interesadas en ejecutar los procesos. La interacción entre la consola de gestión y el cliente puede darse por ejemplo, por medio de una **interfaz web**.

- La arquitectura está pensada para que cada cliente pueda ser una máquina independiente o un servidor de terminales.

1.2.2.5.- Proceso de ejecución.

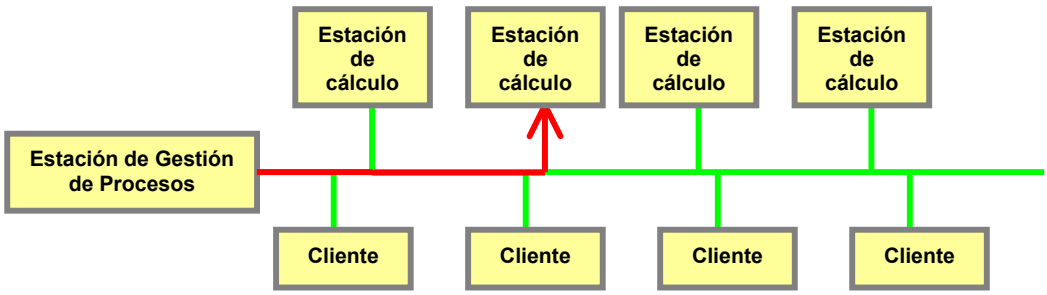
- Primero **se lanza una petición** desde un cliente:



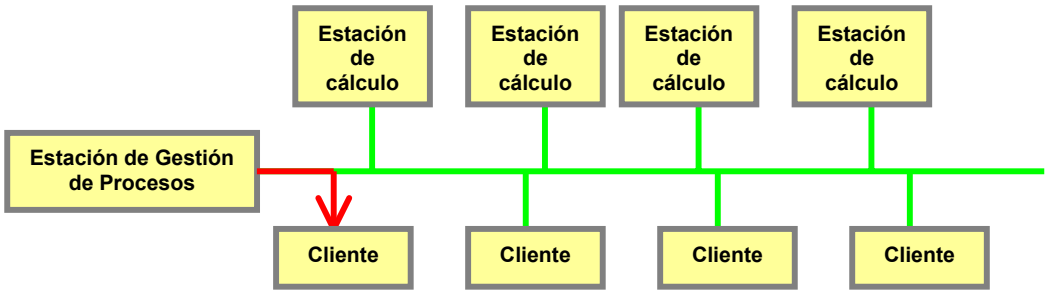
- Después el **Planificador busca en la DB** de procesos si tiene algún dato histórico de ejecuciones de ese proceso en particular. Si no, inserta una nueva entrada en la DB para este proceso, con información sobre tamaño del proceso, fecha de ejecución (puede haber unas determinadas horas donde la carga de procesos es mayor), estimaciones sobre operaciones I/O, etc…

- Ahora **recoge las estadísticas** de las estaciones de cálculo y planifica dónde se puede ejecutar con mayor rapidez el proceso.

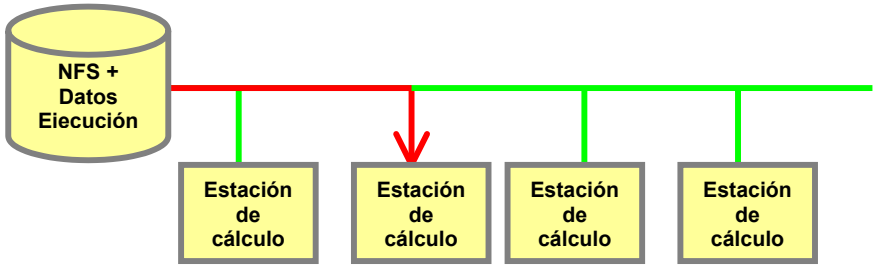
Una vez hecha la planificación, **se lanza el proceso** a la estación correspondiente:



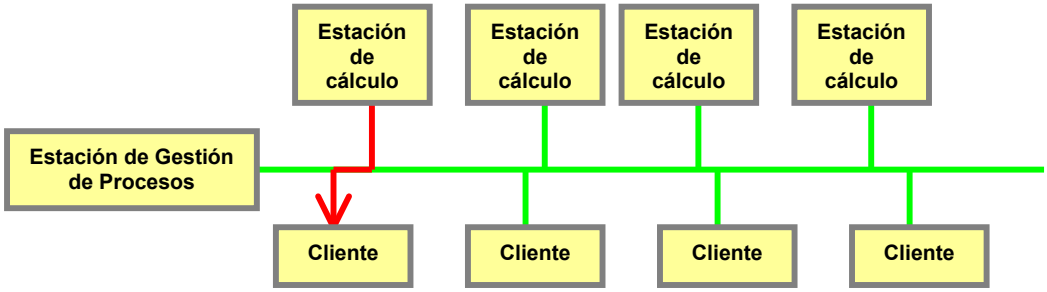
- El planificador le pasa el identificador de la estación de cálculo al cliente, y de esta forma establecen entre los dos una **comunicación independiente.**



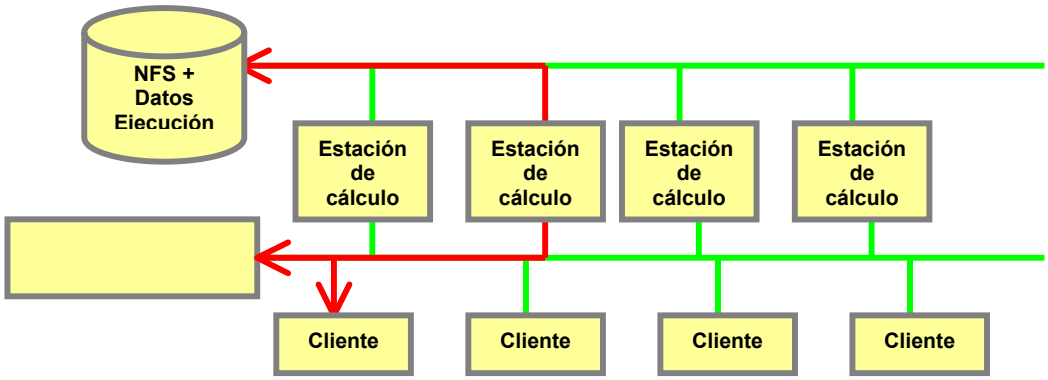
- La **estación de cálculo ejecuta los binarios** que se almacenan en el NFS.



- **El proceso se ejecuta.** Si hay algún problema de espacio, capacidad de proceso, error en la máquina, etc, el agente recolector está a la escucha y le envía los mensajes al planificador, que decide si cambiar el proceso de máquina o abortarlo. Si no hay problemas, la estación de cálculo vuelca el resultado del proceso en el almacenamiento compartido, y avisa al planificador. Si hay salida por pantalla, se redirige directamente al cliente.



- Ahora **la estación de cálculo envía la notificación al cliente y a la estación de gestión** de que el proceso ha terminado y del lugar donde se almacenan los resultados de la ejecución (si es aplicable).



- Finalmente **el planificador recoge los tiempos de ejecución, parámetros de I/O, etc**… del proceso y actualiza el contenido de la DB, que es de utilidad para posteriores ejecuciones del proceso.

1.2.3.- Propuesta detallada: Algoritmos del planificador.

1.2.3.1.- Aspectos generales.

A la hora de asignar tareas a procesadores, podemos buscar la optimización de:

- **Rendimiento**, evitando muchos cambios de contexto (esto favorecería a los procesos largos).

- **Tiempo de respuesta**, ejecutando en primer lugar los procesos cortos.

- **"Justicia máxima"**, aunque esto penalizaría el rendimiento y el tiempo de respuesta, pues obligaría a realizar muchos cambios de contexto.

Dependiendo de la *naturaleza del sistema*, podremos desarrollar los siguientes tipos de algoritmos:

Algoritmos deterministas.

Son adecuados cuando se sabe de antemano todo acerca del comportamiento de los procesos. De este modo, es posible realizar una asignación perfecta. En nuestro caso, toda la información de la que pudiéramos disponer a priori acerca del proceso, podríamos incluirla en la Base de Datos para su posterior consulta.

Pero esto no ocurre en prácticamente ningún sistema conocido. Lo que sí podemos encontrar son aproximaciones razonables. Por ejemplo, en los bancos, el trabajo de un día es similar al del día anterior.

Algoritmos heurísticos.

En caso de no tener datos acerca del comportamiento de los procesos, la carga es por completo impredecible. Necesitamos técnicas heurísticas para hacer las asignaciones.

Algoritmos centralizados.

La recolección de toda la información relevante en un solo lugar permite tomar una mejor decisión, pero es poco robusto y coloca una carga pesada en la máquina central.

Algoritmos distribuidos.

Son preferibles a los anteriores, pero existe carencia de alternativas descentralizadas adecuadas.

1.2.3.- Propuesta detallada: Algoritmos del planificador.

Algoritmos óptimos o subóptimos.

A menudo, el buscar la mejor asignación resulta demasiado caro: recolectar y procesar demasiada información. En la práctica, la mayoría de algoritmos buscan soluciones subóptimas, buscando rentabilizar los recursos utilizados.

Políticas de transferencia locales / globales

Al asignar un proceso ¿tendremos en cuenta la carga local de la máquina, o la carga global del sistema? En general, los algoritmos globales dan un resultado ligeramente mejor, pero a un coste significativamente más caro.

Políticas de ubicación.

En caso de ejecución remota del proceso ¿a qué maquina se asigna?.
En general, el equipo emisor notificaría su necesidad de hallar una máquina remota capaz de tomar el proceso, y a su vez, las máquinas ociosas notificarán su condición de disponibles.

1.2.3.2.- Algoritmos ciegos.

1.2.3.2.1.- Aleatorio.

El sistema se limitará a elegir máquina de forma aleatoria.

1.2.3.2.2.- Turno Rotatorio no expropiativo (estático).

Se asignarán las estaciones por estricto turno. Una vez hecha la asignación, la estación no liberará el proceso hasta que éste termine

1.2.3.2.3.- Turno rotatorio expropiativo (dinámico).

Similar al anterior, pero fijaremos un intervalo relativamente corto de tiempo, llamado “quanto”, al final del cual, la estación liberará el proceso para que sea asignado de nuevo.

1.2.3.2.4.- LRU (Least Recently Used).

Se asignará la estación que lleva inactiva durante más tiempo

1.2.3.2.5.- LFU (Least Frequently Used).

Se asignará la estación que ha sido menos veces utilizada

1.2.3.3.- Algoritmos que consultan la Base de Datos.

1.2.3.3.1.- Por eficiencia de CPU.

Se asignará el proceso al sistema más eficiente que no se encuentre ocupado en ese momento

1.2.3.3.2.- Por eficiencia de ancho de banda.

Será necesario la utilización de una herramienta eficiente para la monitorización de los recursos de red del sistema.

1.2.3.3.3.- Por evaluación de comportamientos pasados.

La base de datos deberá ser capaz de ir guardando los tiempos de respuesta de las estaciones de cálculo según los distintos tipos de procesos (fibonacci, función de ackermann, procesos de E/S, etc.), para poder asignar los procesos a las estaciones que, a priori, han dado mejor resultado. Sin duda, uno de los apartados más elaborados del proyecto.

1.2.3.4.- Algoritmos inteligentes.

1.2.3.4.1.- Cálculo del tiempo de ejecución.

Estos tipos de algoritmos van a resultar muy complicados de implementar, ya que es muy difícil estimar eficientemente el tiempo en que un proceso se ejecuta, si no es consultando evaluaciones anteriores.

Pueden resultar de utilidad algunos algoritmos de planificación en monoprocesadores, como SPN (Shortest Process Next), que realiza una estimación del tiempo restante de ejecución de un proceso mediante la siguiente fórmula:

1.2.3.- Propuesta detallada: Algoritmos del planificador.

$$S_{n+1} = \frac{1}{n}\sum_{i=0}^{n} T_i$$

Donde:
T_i = Tiempo de ejecución total del i-ésimo proceso
S_i = Valor pronosticado para el caso i-ésimo

Para evitar recalcular la suma completa cada vez, podemos reescribir esta ecuación de la siguiente manera:

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n$$

No obstante, esta fórmula da la misma importancia a todos los procesos, y, normalmente es conveniente dar más peso a los casos más recientes. Para ello, utilizamos lo que llamamos "**promediador exponencial**":

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n \qquad 0 < \alpha < 1$$

Cuanto mayor sea α, mayor importancia daremos a los últimos procesos frente a los anteriores. Para verlo con más claridad, consideremos el desarrollo de la ecuación anterior:

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_0$$

1.2.3.4.2.- Algoritmos basados en "Token Ring"

Basado en el protocolo de red, a la hora de asignar un proceso a una estación, se creará un testigo, que se irán pasando las estaciones. Cada una marcará en el **testigo** su **prioridad** de toma del proceso. Una vez que el testigo haya visitado todas las estaciones, la estación que haya marcado la mayor prioridad tomará el proceso. Se puede hacer la versión expropiativa y no expropiativa de este algoritmo.

Los problemas más relevantes que puedan presentar este tipo de algoritmos estarán relacionados con la robustez y la escalabilidad.

1.2.3.4.3.- Grupos de procesos.

Este tipo de algoritmos puede ser uno de los más importantes. Se basa en la búsqueda de aquellos procesos relacionados entre sí para que sean **despachados simultáneamente**, y de esta manera, evitar esperas de resultados, recepción de mensajes, etc. Además, procuraremos que se ejecuten de forma que la comunicación que se tenga que realizar entre ellos sea sencilla.

1.2.4.- Puntos críticos.

1.2.4.1.- Problemas del Shell.

El principal problema del proyecto es crear o modificar el shell que utilizará el usuario para ejecutar los procesos. Dicho shell debería comportarse igual que un shell estándar de Unix, por ejemplo bash.

La alta complejidad de esta parte reside en tratar las características más avanzadas del mismo:

- **Background, Foreground**:
  Este problema se podría tratar con una conexión TCP entre el equipo procesador y el cliente que intercambiara la información. En el caso del Background simplemente se desecharían los datos que provienen del procesador.

- **Interrupciones** (Crtl+C, Crtl+D, ...):
  Las interrupciones habría que capturarlas en el cliente, identificando el proceso al que van dirigidas. A partir de ahí y como existe en todo momento una conexión cliente-procesador para la entrada/salida de datos, utilizar esa conexión para enviar la interrupción al procesador.

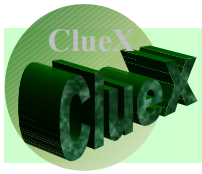
- **Pipes**:
  Nos referimos aquí al intercambio de datos entre dos procesos (en el caso de proceso-archivo no habría problema). Para solucionarlo habría que crear una conexión TCP entre los dos procesos (suponiéndolos en máquinas distintas) y redirigir la información por ese canal. En el caso de que estuvieran en la misma máquina, podría usarse cualquier mecanismo IPC estándar de Unix para solucionar el problema.

1.2.4.2.- Diferenciación de procesos.

Otra parte crítica sería la diferenciación entre **procesos de consola** (modo texto) y **aplicaciones gráficas** que utilicen el servidor X de ventanas, ya que la comunicación entre el procesador y el cliente variaría dependiendo del tipo; en el primer caso utilizaríamos una conexión TCP para enviar el texto (redirigiendo los identificadores de entrada/salida estándar del proceso) y en el segundo se utilizaría el mecanismo ya implementado del modelo cliente/servidor de las X.

En este segundo caso hay que indicar al proceso, por medio de un parámetro, en qué dirección IP ha de mostrar los datos. La IP en cualquier caso ha de ser conocida, pero la forma de indicárselo al proceso varía de una aplicación a otra.

Una posible solución podría ser tener una base de datos en el planificador que diferenciara si un proceso es de texto o gráfico. Si se lanzara un proceso el cual no estuviera en dicha base de datos habría que preguntar al usuario.



Análogamente, en el caso de las aplicaciones gráficas, también habría que guardar de qué forma se indica al proceso la **dirección IP del cliente X** donde mostrar los gráficos. De esta forma el planificador añadiría automáticamente este parámetro antes de enviar el proceso a un equipo de cálculo. Otra vez aparece el problema de qué hacer cuando el proceso no aparece en la base de datos, pero en este caso se podría intentar probar con los parámetros típicos (en casi todos los comandos para las X el display se indica de forma similar) antes de preguntar al usuario.

1.2.4.3.- Mecanismos IPC.

Surge otro problema cuando procesos que se estén ejecutando en máquinas distintas intenten utilizar mecanismos IPC del Unix System V. Este problema no tiene solución a no ser que se modificara radicalmente el Kernel, pero eso resulta inviable en este proyecto. La única **solución parcial** posible es que los equipos de cálculo detecten que el proceso ha fallado por ese motivo, y lo comuniquen al planificador para que lo vuelva a ejecutar en el mismo equipo donde esta el otro proceso con quien ha de comunicarse. El planficador guardaría esa información en su base de datos para futuras ejecuciones de ambos procesos.

La comunicación mediante tuberías no debería dar ningún problema al efectuarse utilizando el sistema de ficheros.

1.2.4.4.- Creación de procesos en los equipos de cálculo.

Puede surgir que los procesos que estén ejecutándose en los equipos de cálculo realicen llamadas *fork()* para crear nuevos procesos. Esta situación escaparía fuera del control del planificador y podría **falsear las estadísticas** de ejecución de los procesos. Por otra parte, esos nuevos procesos necesitarán comunicarse con el usuario para la entrada/salida de datos, pero en principio no representa gran problema ya que si se redirigen los descriptores de entrada/salida/error del padre hacia una conexión de red, el hijo debería heredarlos.

El problema de la incongruencia de los datos del planificador podría resolverse si los equipos de cálculo **informan periódicamente de los procesos que ejecutan**. Como estos equipos han de informar del estado de CPU, memoria, ..., se podría aprovechar ese mismo paquete de red para enviar la información acerca de los procesos.

1.2.4.5.- Cuellos de botella.

Algo ineludible en todo el diseño son los cuellos de botella debidos a la falta de **capacidad de la red, del planificador o del servidor del sistema de ficheros.**

El primer caso depende totalmente de la tecnología empleada, pudiendo ser sustituida por otra de mayor capacidad si fuera necesario.

En el caso de que el planificador o el servidor de ficheros no tuvieran capacidad suficiente, la solución sería replicar esos equipos dividiendo el trabajo.

1.2.4.6.- Tolerancia a fallos.

Hay que tener en cuenta los posibles fallos por **caídas de los equipos de la red**. En el caso de los equipos de cálculo se podría detectar fácilmente porque han de informar periódicamente de su estado al planificador. Si este no recibe paquetes de estado con un determinado retardo, eliminaría ese equipo de las planificaciones y relanzaría sus procesos.

En el caso del planificador y el sistema de ficheros la única solución sería la redundancia ya que por la topología de la red ningún otro equipo podría sustituirles.

1.2.4.7.- Seguridad y Confidencialidad.

En el aspecto de la identificación de usuarios y propietarios de los ficheros no aparece ningún inconveniente ya que se utilizarían los mecanismos estándar de todos los Unix. El problema puede aparecer con la confidencialidad de los datos que circulan por la red.

Una solución podría ser utilizar sistemas de encriptación como **SSH,** aunque también hay que tener en cuenta que está basado en criptografía de clave publica/privada la cual puede incorporar grandes retardos. Si se comprueba experimentalmente que el retardo es excesivo siempre se podría implementar un sistema de encriptación propio. Éste podría consistir en un sistema de Certificación para comprobar la identidad de los equipos y a partir de ahí generar una **clave simétrica** que a través de un algoritmo sencillo encriptará los datos para la conexión en curso.

1.2.4.8.- Planificador eficiente.

Como sabemos los algoritmos de IA consumen mucho tiempo de CPU y memoria. Debemos llegar a un compromiso tal que nuestros algoritmos tengan mejores decisiones que un simple FIFO, y no tarden mucho más que éste, es decir, el tiempo de decisión debe afectar mínimamente al tiempo global de ejecución del proceso.

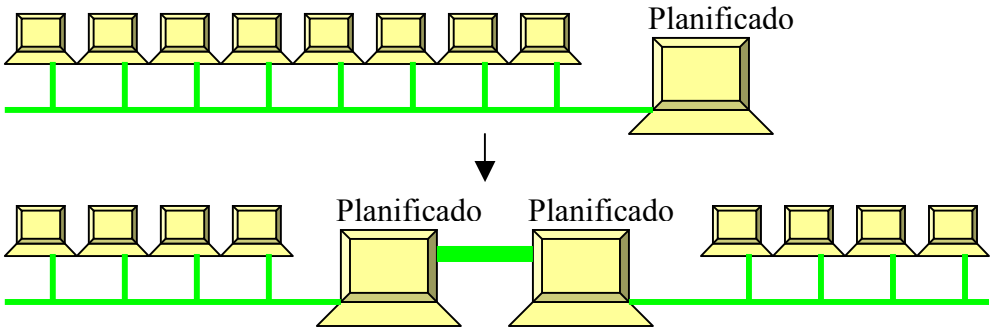
1.2.4.9.- Aprendizaje, Data Mining y algoritmos de decisión.

Implementar un algoritmo que realmente sepa inferir carga de proceso en el futuro, que relacione unos procesos con otros, y que tenga funciones heurísticas y pesos asociados a cada valor calculado que sean correctas para una decisión acertada del planificador.

1.2.4.10.- Flexibilidad y facilidad de expansión

Respeto a la facilidad de **incorporación de nuevos equipos** al cluster (excluimos los clientes), hay que prever que la conexión de muchos equipos de cálculo a una misma red puede crear grandes retrasos en ella (sobre todo si utilizamos tecnología Ethernet). Por eso habría que agrupar un máximo de equipos de cálculo con un planificador y hacer de esta forma varios grupos con los equipos. Cada grupo de equipos estaría conectado por su propia red independiente para solucionar así el problema de los retardos. Con esta nueva arquitectura los clientes que se conecten al planificador por defecto, podrán ser redirigidos a otro planificador que esté en otro grupo. Por otro lado los planificadores necesitarán comunicarse entre sí para compartir las estadísticas.

Como todo este proceso es dependiente de la interconexión física de los equipos tendrá que ser configurado manualmente por el administrador.



1.2.5.-Propuesta detallada: Propuestas de arquitectura.

1.2.5.1.- Introducción.

En este apartado pretendemos dar a entender la evolución de la propuesta de la arquitectura final de ClueX.

En primer lugar, veremos la **propuesta 1**, que organiza el cluster de una forma parecida a lo que sería el **Token Ring** de los protocolos de red. Esta propuesta fue rechazada porque el tráfico de red sería excesivo, ya que al estar la decisión descentralizada (repartida entre todos los miembros del cluster) y tener que ir pasando por todas las estaciones de cálculo para decidir en cuál se va a ejecutar el proceso actual, el retardo de decisión es importante, y además crece cuantas más estaciones de cálculo pongamos. Se hizo una segunda aproximación es esta propuesta de no utilizar estructura física de anillo, y usar broadcast, pero eso puede saturar las estaciones ocupadas con mensajes de red que no les interesan. Es decir, **la arquitectura no es escalable.**

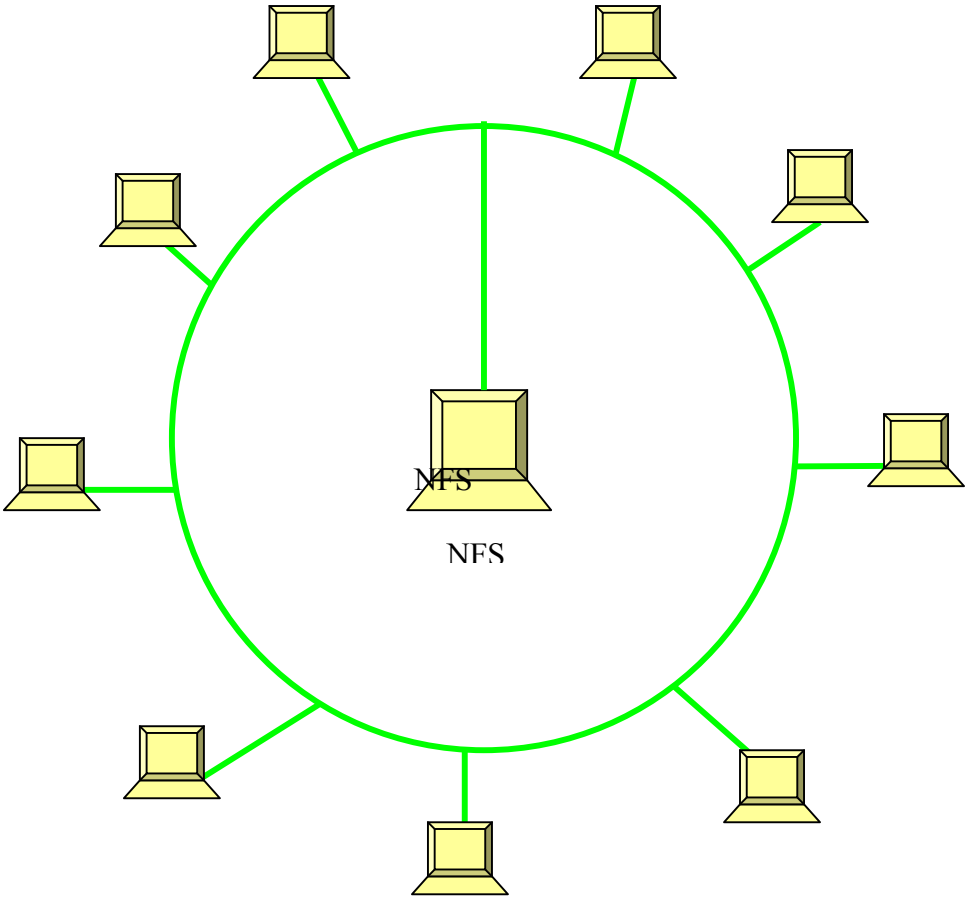
En la **propuesta 2**, se presentan ideas interesantes para la descentralización de la planificación, lo que hace del cluster un sistema robusto, pero vimos que fundiendo cliente con estación de proceso, no vamos a obtener nunca un servicio de escalabilidad, ya que, aunque se aprovecha la potencia de cálculo del cliente, como todos los clientes van a querer ejecutar un proceso, al final la mejor decisión será ejecutarlo siempre en la propia máquina.

En la **propuesta 3**, se proponen dos redes independientes para clientes y estaciones de cálculo. En esta propuesta falla que la comunicación de cliente con estación de cálculo sólo se hace a través del planificador, lo que sobrecarga demasiado a esta máquina en cuanto a tráfico de red. También se puede ver que el poner el NFS compartido entre clientes y estaciones de cálculo no optimiza el tráfico de red. Es la propuesta más aproximada a la decidida para ClueX, y, por ello, es explicada con mayor detalle.

Por ello se llegó a la arquitectura que hemos propuesto, planificación **centralizada pero desasistida**, en cuanto a que cuando se planifica el proceso, el planificador se "olvida" de éste hasta que termina, y la comunicación cliente-estación de cálculo es privada, con lo que la red no se satura.

1.2.5.2.- Propuesta 1: Variante del "Token Ring"

La siguiente arquitectura pretende ser **lo más distribuida posible**, evitando la sobrecarga de alguna de las estaciones y cuellos de botella en el tráfico. Un esquema aproximado sería:



Las **estaciones funcionarían tanto como clientes como estaciones de cálculo**, y estarán organizados en un **anillo lógico**, pero sólo formarían parte de él aquellas estaciones que puedan aceptar carga adicional. Se implementará la forma de añadir o separar una estación del anillo.

Los datos necesarios para saber qué máquina se comportará mejor ante la asignación de un proceso de un tipo determinado estarán **almacenados de forma local** en cada máquina. El tratamiento de esta información se realizaría de la siguiente manera:

Cada estación, de acuerdo con los datos de que dispone en relación con comportamientos anteriores, datos de E/S, rendimiento, carga actual de trabajo, etc., tendría un **indicador numérico** con el que expresaría su **estimación de calidad** de servicio que daría a un proceso de cada tipo determinado.

1.2.5.-Propuesta detallada: Propuestas de arquitectura.

La asignación de un proceso se gestionaría mediante el envío por parte del cliente de un **"testigo", que recorrería el anillo** con información acerca de la naturaleza del proceso, y con datos acerca del "candidato" más prometedor para hacerse con el proceso. A su paso por una estación, ésta comprobaría con qué calidad daría servicio a un proceso de tal tipo. Si el indicador que tiene al respecto mejora al que tiene el testigo, actualizaría la información del mismo. Después, pasa el testigo a la siguiente estación.

A la llegada del testigo al cliente, éste asignará el proceso a la estación que le indique.

Esta arquitectura ofrece muchas **variantes**: por ejemplo, la posibilidad de enviar testigos alternativamente en un sentido u otro, o fijar un **valor "umbral"** de calidad con la que queremos despachar el proceso: en tal caso, al hallar la primera máquina que, con su dato de calidad de servicio del proceso, superase el umbral, ya no sería necesario terminar el recorrido del anillo, y se realizaría directamente la asignación del proceso. Del mismo modo, se podría fijar una **"cota inferior":** si ninguna estación nos llegara a ofrecer ese mínimo de calidad de servicio esperada, a la llegada del testigo al cliente no se realizaría la asignación del proceso, sino que permanecería en cola de espera y se volvería a lanzar otro testigo después de un cierto tiempo (con ayuda de un temporizador).

Como ya se ha mencionado, la principal ventaja de esta arquitectura es la facilidad con la **evita congestiones y cuellos de botella** en el tráfico de red. Además, el hecho de que la información de planificación resida en bases de datos locales a las máquinas, permite que dicha información sea fácil de tratar, y simplifica en gran medida los algoritmos de planificación.

Pero también encontramos desventajas muy a tener en cuenta. Las principales son la aparente **falta de robustez** del sistema por tratarse de un anillo (a pesar de que sea un anillo lógico y tengamos mecanismos eficaces para meter y sacar estaciones del mismo), y, sobre todo, las **dificultades de implementar esta arquitectura en grandes sistemas.**

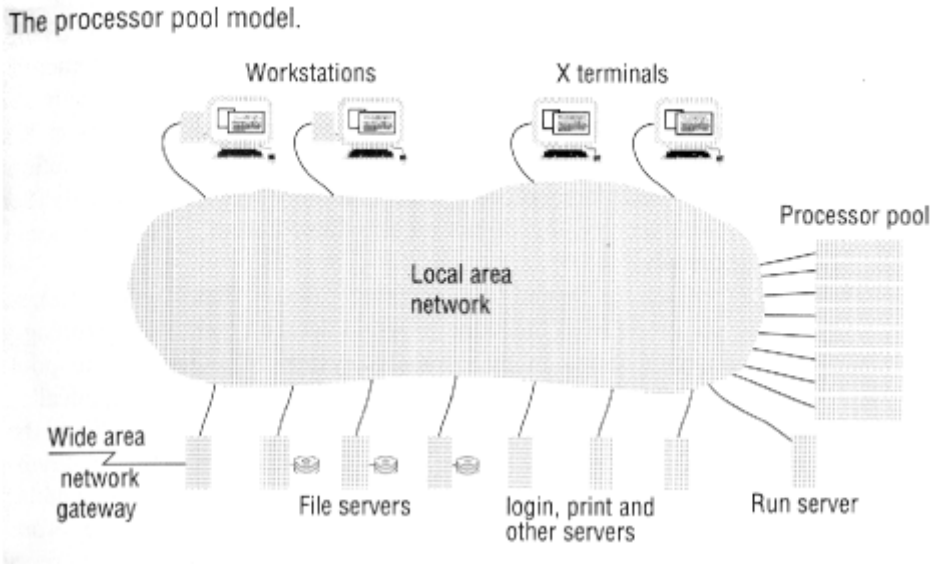
1.2.5.3.- Propuesta 2: Cliente y estación de cálculo integrados.

En esta propuesta de arquitectura se pretende **descentralizar** la planificación de los procesos de un único equipo.

La opción centralizada o *processor pool model* se compone una LAN formada por clientes separados de los procesadores (*processors pool*). El uso de los procesadores esta controlado por un planificador (*run server*).

En nuestro caso el planificador destacaría por su capacidad de recoger estadísticas acerca del estado de los procesadores y los recursos que han necesitado los procesos ejecutados para poder **mejorar la planificación** a partir de la experiencia.

Aquí podemos ver un esquema:



Distributed Systems: Concepts and Design. Coulouris, Dollimore y Kindberg.

Uno de los problemas más importantes que plantea esta arquitectura es la necesidad de actualizar el estado del planificador con información acerca de la ejecución de los procesos y los procesadores. Si la LAN que interconecta los ordenadores no posee un ancho de banda considerable, puede ocasionar un **cuello de botella** que haga bajar el rendimiento.

En esta arquitectura Cliente-Planificador-Procesador separados, tenemos al menos seis transacciones de red para ejecutar un proceso:

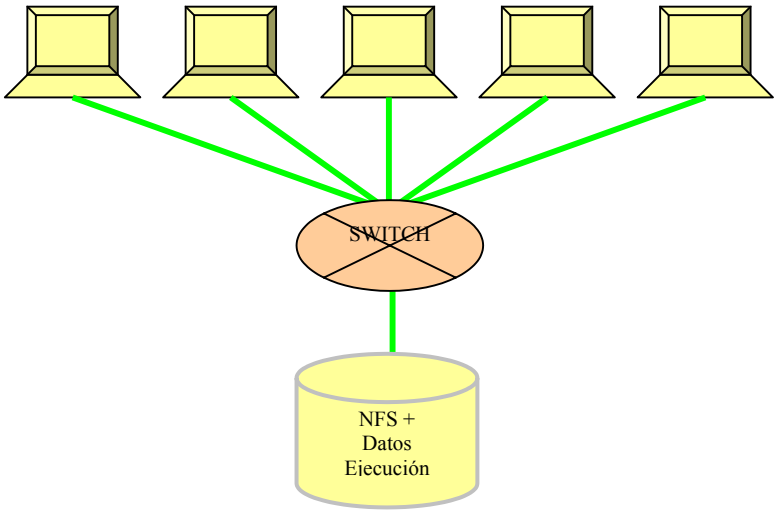
- Cliente - Planificador.
- Planificador (petición de estado de CPU, memoria, ...) - Procesadores.
- Procesadores (estadísticas de CPU, memoria, ...) - Planificador.
- Planificador - Procesador.
- Procesador - Cliente.
- Procesador (estadísticas del proceso) - Planificador.

Otra objeción importante es que para una red doméstica, es decir unos pocos ordenadores, es **inviable diferenciar entre clientes, planificador y estaciones de cálculo.**

Para intentar paliar estas restricciones podemos proponer un modelo en el que **fundamos estas tres funcionalidades en cada uno de los equipos**. Es decir, cada ordenador se encargará de:

- ✓ Funcionar como cliente cara al usuario.
- ✓ Planificar los procesos que el usuario ejecute desde esa máquina.
- ✓ Recoger las estadísticas de ejecución de los procesos.

El esquema de conexión sería el siguiente:



En éste podemos diferenciar claramente, que el único elemento que comparten los equipos son el espacio de almacenamiento, que por un lado incluye los archivos propios del S.O. y por otro los datos (estadísticas) de ejecución de los procesos.

Respecto al funcionamiento, podríamos decir que cada equipo se comporta de forma independiente a los demás, pero colaborando con ellos para poder llevar a cabo sus objetivos de la mejor forma posible (**Arquitectura de Agentes Software**).

El funcionamiento de este nuevo sistema sería:

1. **En cada equipo se ejecuta un cliente** para que el usuario pueda mandar a ejecución sus procesos. Cuando se une el nodo a la red por primera vez, se leen de la base de datos las características de los otros equipos.

2. **Periódicamente** (podríamos fijar un tiempo aproximado de 2 a 5 segundos dependiendo de la congestión de la red), **cada equipo publica** a toda la red **su estado**: CPU, memoria, ...

3. **Cuando el usuario ejecuta un proceso**, el equipo cliente (que ahora ejerce de planificador) **consulta la base de datos** para obtener los datos de ejecución anteriores de ese proceso.

4. Una vez consultados los datos del proceso, sólo faltaría conocer el estado del resto de equipos para decidir donde ejecutar el proceso. Pero como los equipos publican periódicamente su estado, podemos realizar una estimación de cual será su estado actual (comprobando su evolución en el tiempo hasta ahora) **sin necesidad de hacer más transacciones en la red.**
   Ésto nos ahorra por un lado la petición a los procesadores y su respuesta, lo cual descongestiona la red. Por otro, evita que el planificador tenga que estar esperando a que todos los equipos respondan para decidir qué hacer con el proceso.

5. Cuando el planificador ha elegido uno de los equipos, **manda el proceso** a este, el cual lo ejecuta.

6. Al equipo que ha ejecutado el proceso sólo le quedará **escribir las estadísticas en la base de datos**.

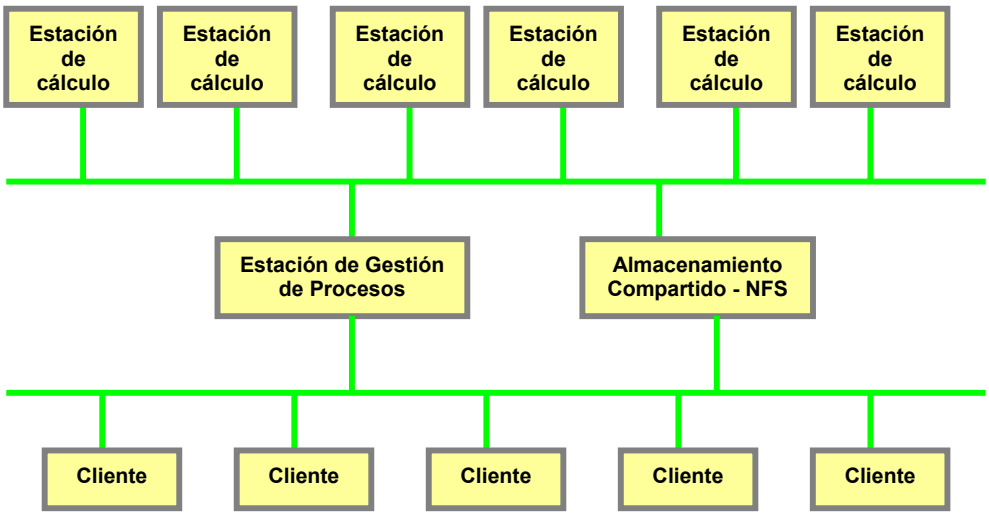
Las características más importantes a destacar en esta arquitectura son:

- ✓ Descontando los paquetes que los equipos envían para publicar su estado, sólo necesitaríamos 2 envíos de red para toda la ejecución de un proceso.
- ✓ Todos los equipos ejercen de planificador, distribuyendo así ese cálculo.
- ✓ No existe ningún cuello de botella en la red. Excepto el nodo donde está el sistema de ficheros, pero podríamos evitarlo dividiendo la base de datos entre varios nodos.
- ✓ Modularizamos más el sistema, al tener el cliente-planificador-procesador en el mismo nodo, y de forma muy independiente a los demás.
- ✓ El desacoplamiento entre los nodos que forman la red aumenta la tolerancia a fallos del sistema.

Los aspectos teóricos que se desarrollarían con este planteamiento estarían relacionados con:

- **Comunicación en Red.**
- **Agentes Software**: planificación y comunicación entre nodos.
- **Razonamiento con incertidumbre**: planificación sin conocer el estado exacto de los otros equipos.
- **Aprendizaje Automático**: mejora continua de la función de planificación basándose en la experiencia.

1.2.5.4.- Propuesta 3: Dos redes independientes



1.2.5.4.1.- Estación de Gestión de procesos.

- Hace el papel de **iniciador de los procesos.**
- Casi todo el código fuente del gestor de procesos se encuentra aquí.
- Consta de varios componentes:

- **Agente recolector de estadísticas**: Escucha en la red los mensajes que lleguen de las estaciones de cálculo, que contienen información útil para las tareas de planificación.

- **Planificador:** Usa los mensajes procedentes de las estaciones de cálculo y su propia base de datos de procesos para inferir cuál será la mejor elección a la hora de lanzar el próximo proceso. Se basará en **parámetros** como carga de CPU, memoria disponible, tamaño del proceso, historia del proceso, I/O involucrada en el proceso, etc… **que guarda en una Base de Datos en la misma máquina**. Se puede definir como un **Agente Software que monitoriza los recursos del cluster y reparte los procesos según convenga en cada momento.** También se encarga de mover procesos entre estaciones en el caso de que un error ocurra, o falten recursos.

- **Consola de gestión:** Aplicación para la gestión del planificador, ajuste de parámetros, etc… También incluye el interfaz de comunicación con los clientes.

1.2.5.4.2.- Almacenamiento compartido.

- El resultado de los procesos debe guardarse en un lugar común, pues no tiene sentido ir a buscarlo a cada una de las estaciones de cálculo.

- **Lo ideal**: tener una máquina enteramente ocupada con esta tarea.

- **Alternativa:** Fusionar esta funcionalidad en la máquina que hace de estación de gestión.

- **Opciones de rendimiento:** Lo ideal sería tener un sistema RAID 5, que nos da alta disponibilidad y alto rendimiento, pero si pensamos en usar máquinas PC debemos pensar en que es más práctico usar RAID 0, que da alto rendimiento y no alta disponibilidad, ya que nuestro objetivo es mejorar el tiempo de ejecución del sistema multiprocesador, pues para usar RAID 5 deberíamos tener al menos 3 discos. EL no tener alta disponibilidad no afecta demasiado ya que éste no es un sistema en producción.

- **Alternativas de arquitectura:** RAID Hardware (lo ideal) o Software.

1.2.5.4.3.- Estaciones de cálculo.

- Aquí se llevan a cabo las tareas reales de proceso.

- **Objetivo:** Escalar en una relación razonable el tiempo de ejecución de un conjunto de procesos que por sí solo satura la capacidad de proceso de una sola máquina.

- **Semejanzas en la arquitectura**: Pensamos que esta arquitectura debe intentar imitar en lo posible un cluster en modo Scalable y/o un software de Application Server.

- En cada estación de cálculo instalaremos un **agente de tamaño reducido** que se limite a **recoger información** de la máquina para pasarla al planificador de la Estación de Gestión. Un protocolo interesante para esta comunicación sería SNMP (Simple Network Management Protocol).

1.2.5.4.4.- Clientes.

- Simplemente son las máquinas que van a aprovechar la capacidad de cálculo del cluster, ya que son las interesadas en ejecutar los procesos. La interacción entre la consola de gestión y el cliente puede darse por ejemplo, por medio de una **interfaz web.**

1.2.5.4.5.- Proceso de ejecución.

- Primero se **lanza una petición** desde un cliente:

- Después el **Planificador busca en la DB de procesos** si tiene algún dato histórico de ejecuciones de ese proceso en particular. Si no, inserta una nueva entrada en la DB para este proceso, con información sobre tamaño del proceso, fecha de ejecución (puede haber unas determinadas horas donde la carga de procesos es mayor), estimaciones sobre operaciones I/O, etc…

- Ahora **recoge las estadísticas** de las estaciones de cálculo y planifica dónde se puede ejecutar con mayor rapidez el proceso.

- Una vez hecha la planificación, **se lanza el proceso a la estación correspondiente**. La comunicación de entrada salida por pantalla se realiza a través del planificador, que actúa como gateway.

- **El proceso se ejecuta**. Si hay algún problema de espacio, capacidad de proceso, error en la máquina, etc, el agente recolector está a la escucha y le envía los mensajes al planificador, que decide si cambiar el proceso de máquina o abortarlo. Si no hay problemas, la estación de cálculo vuelca el resultado del proceso en el almacenamiento compartido, y avisa al planificador.

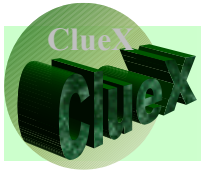
- Por último la consola de gestión envía la **notificación al cliente de que el proceso ha terminado** y del lugar donde se almacenan lor resultados de la ejecución (si es aplicable).

MÓDULO II

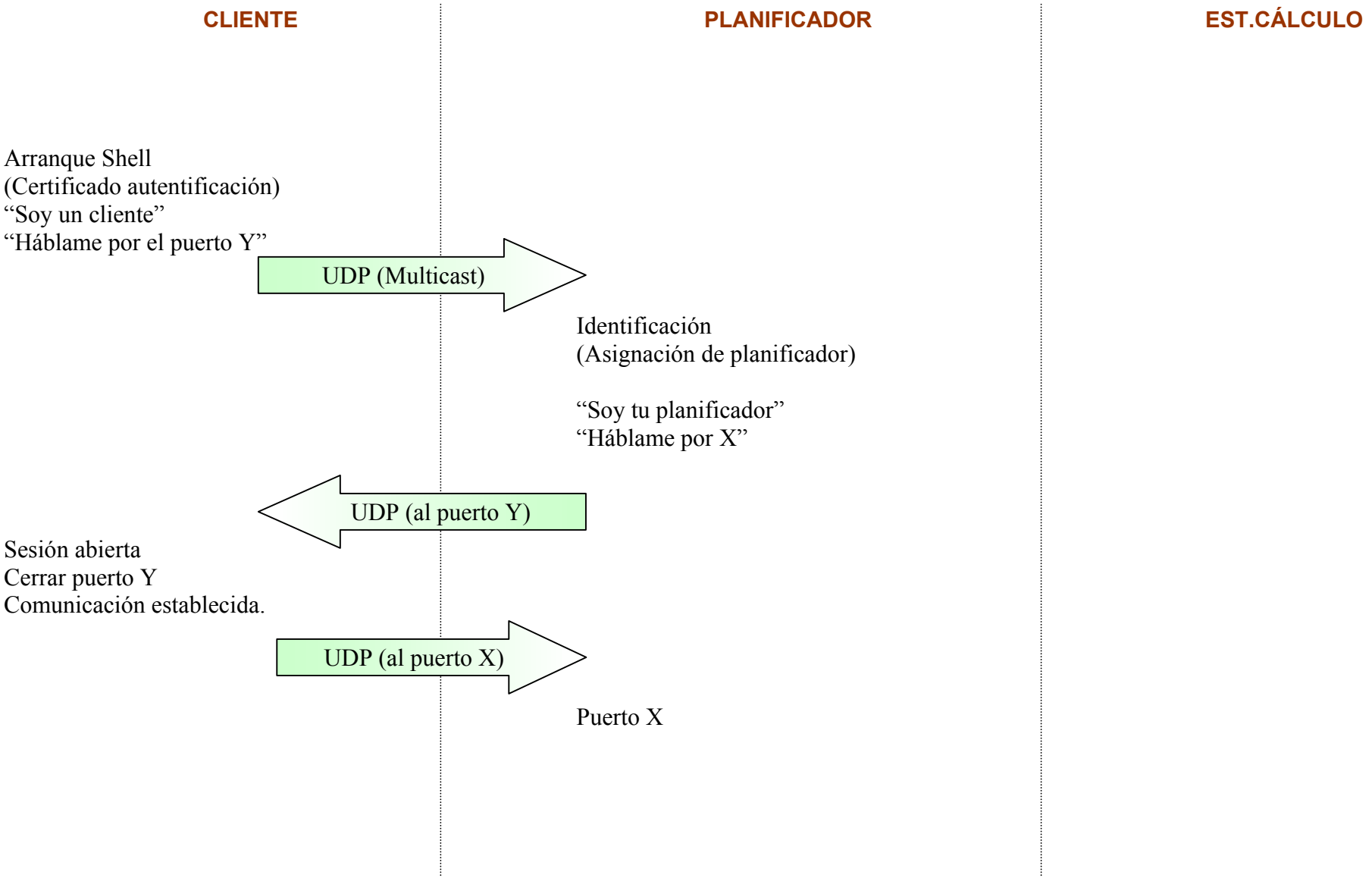
ARQUITECTURA

DEL

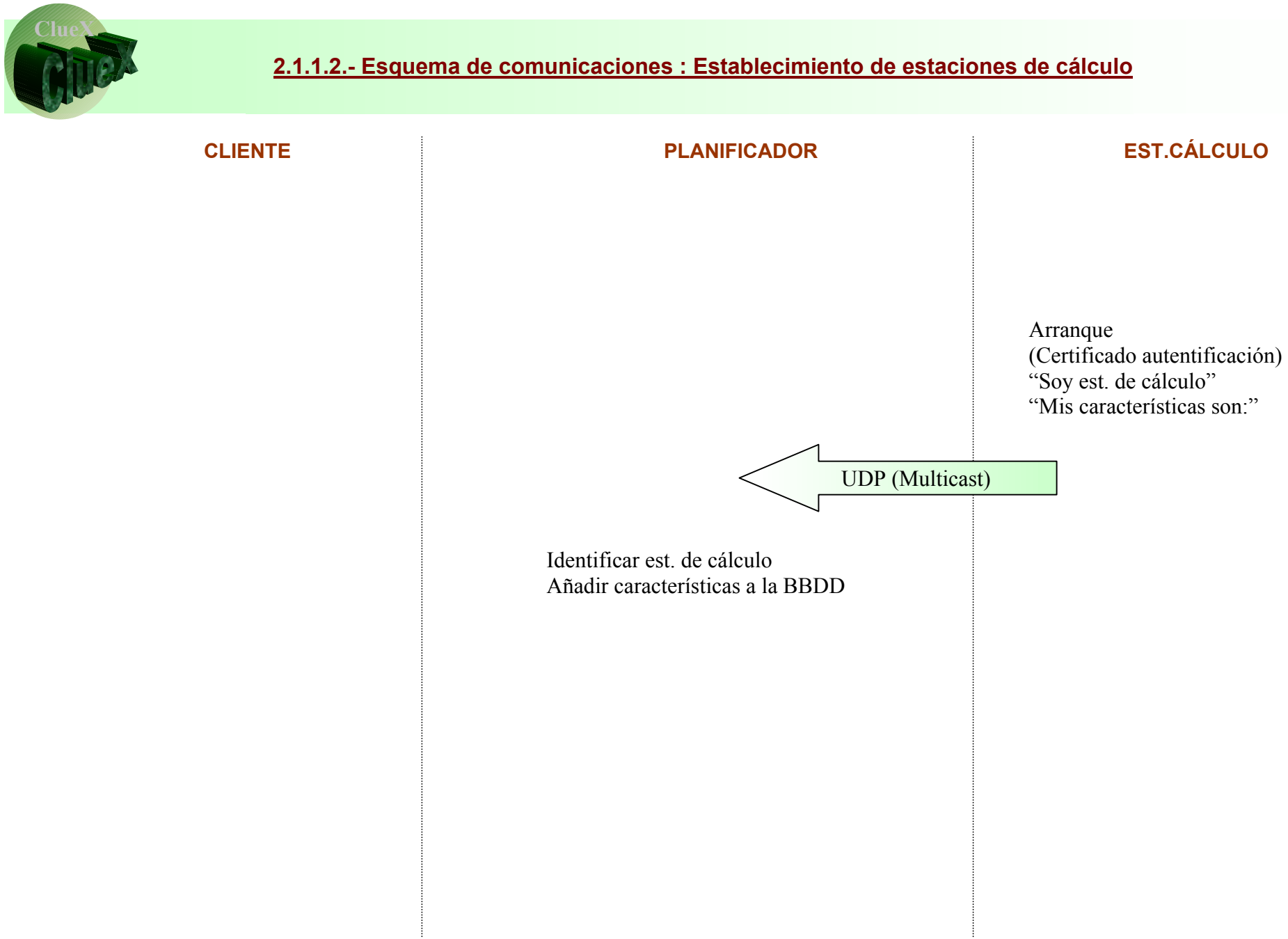
SISTEMA

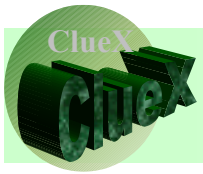


2.1.1.1.- Esquema de comunicaciones: Establecimiento del cliente

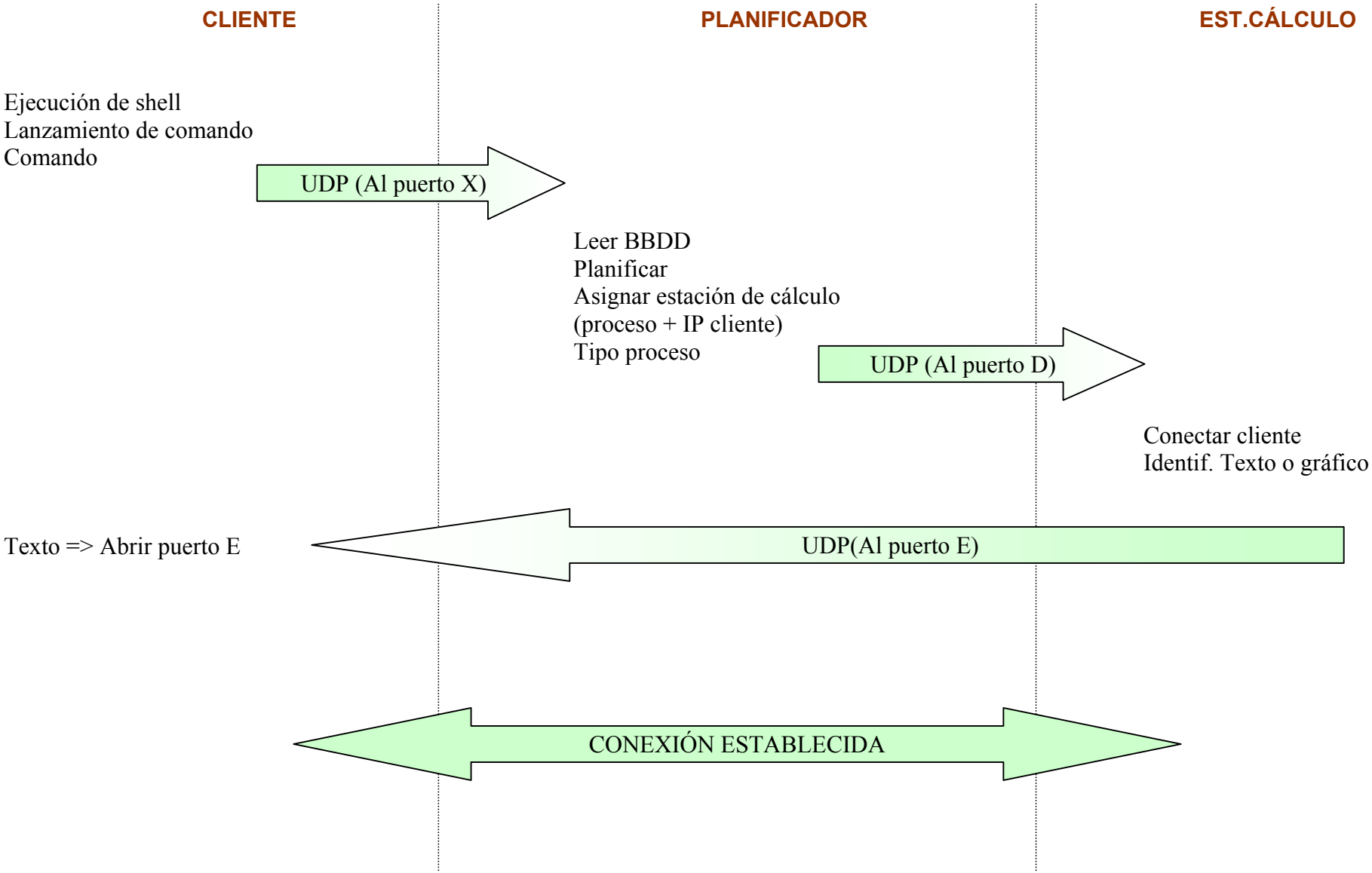


2.1.1.2.- Esquema de comunicaciones : Establecimiento de estaciones de cálculo

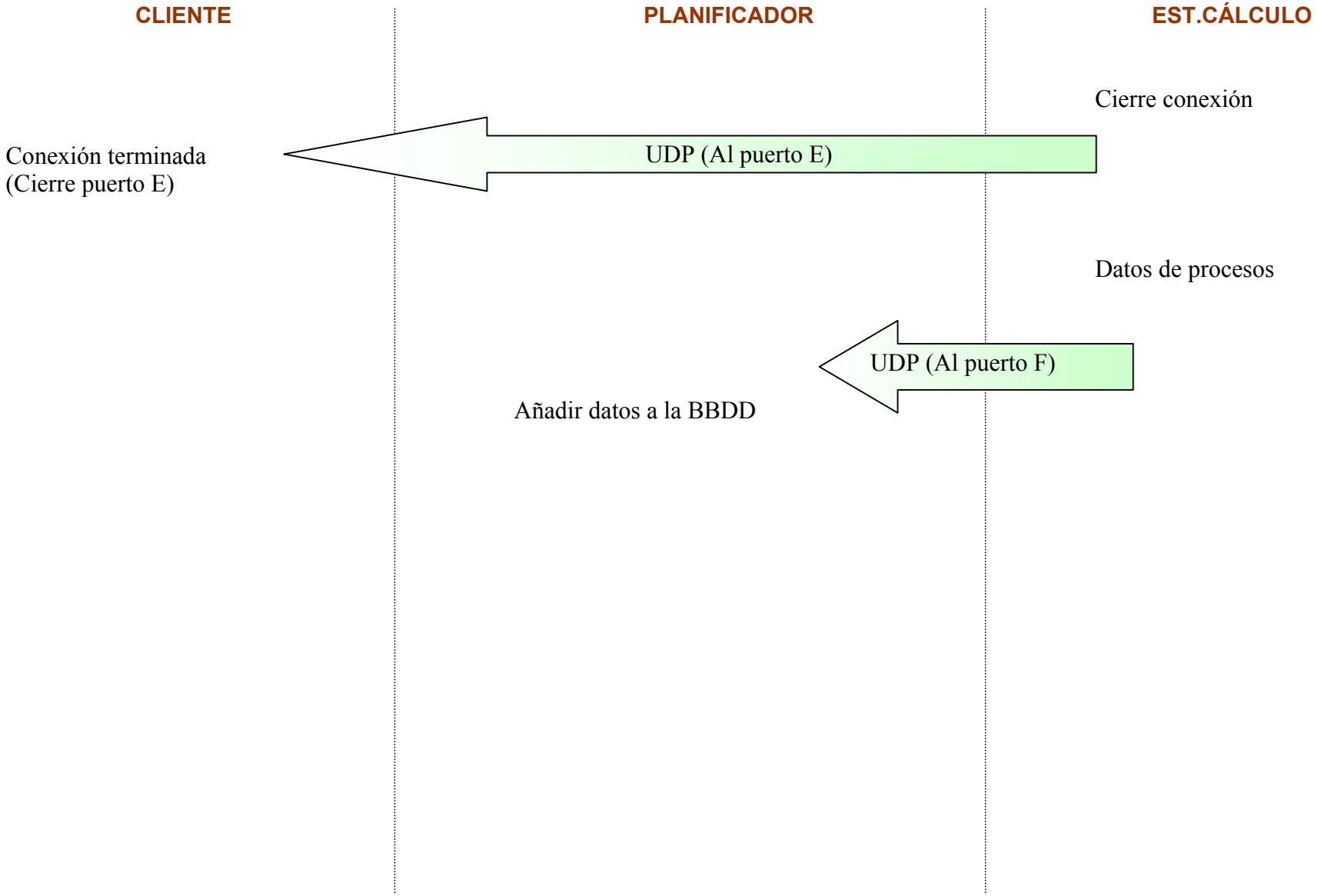




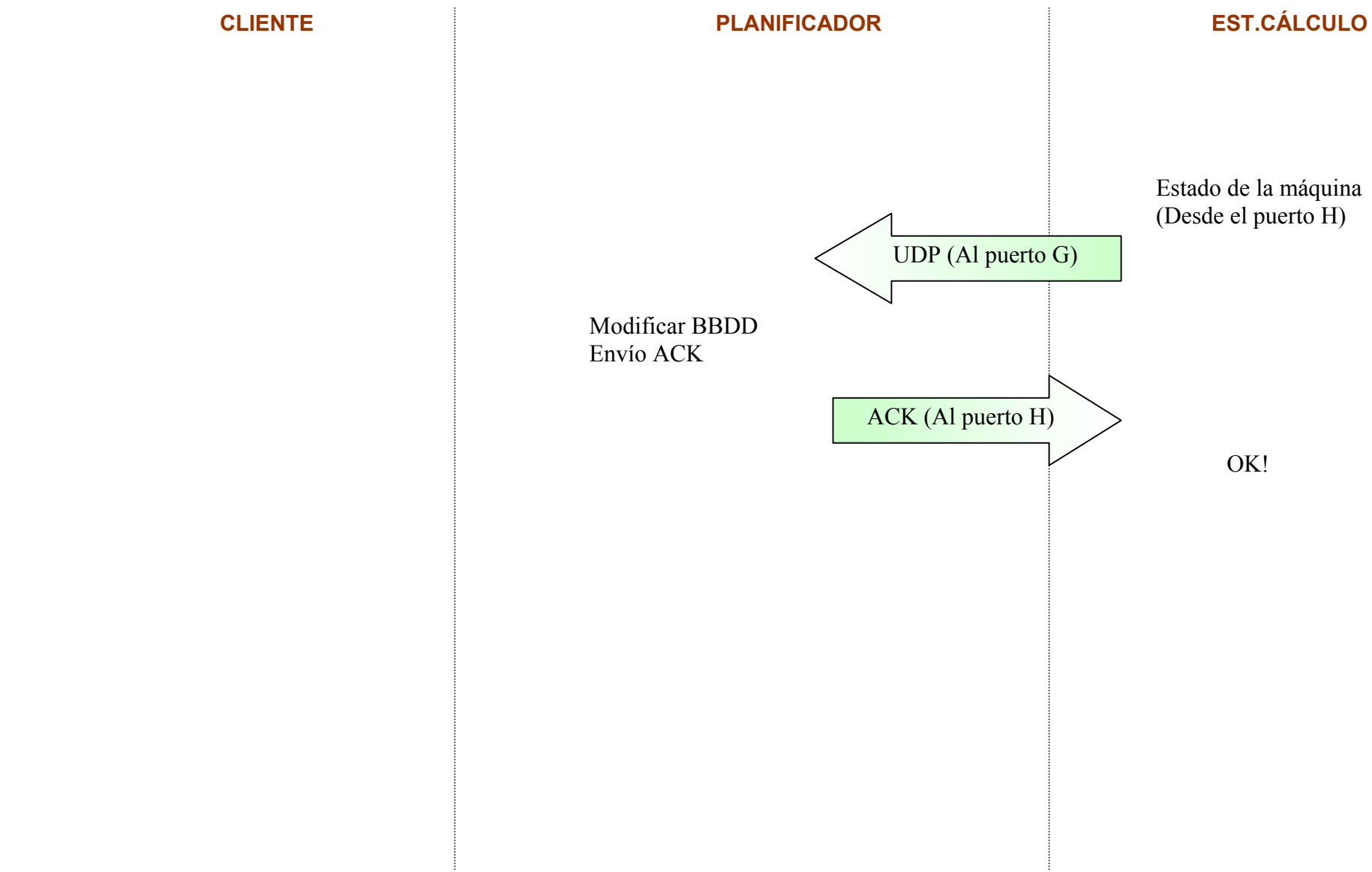
2.1.1.3.- Esquema de comunicaciones: Funcionamiento



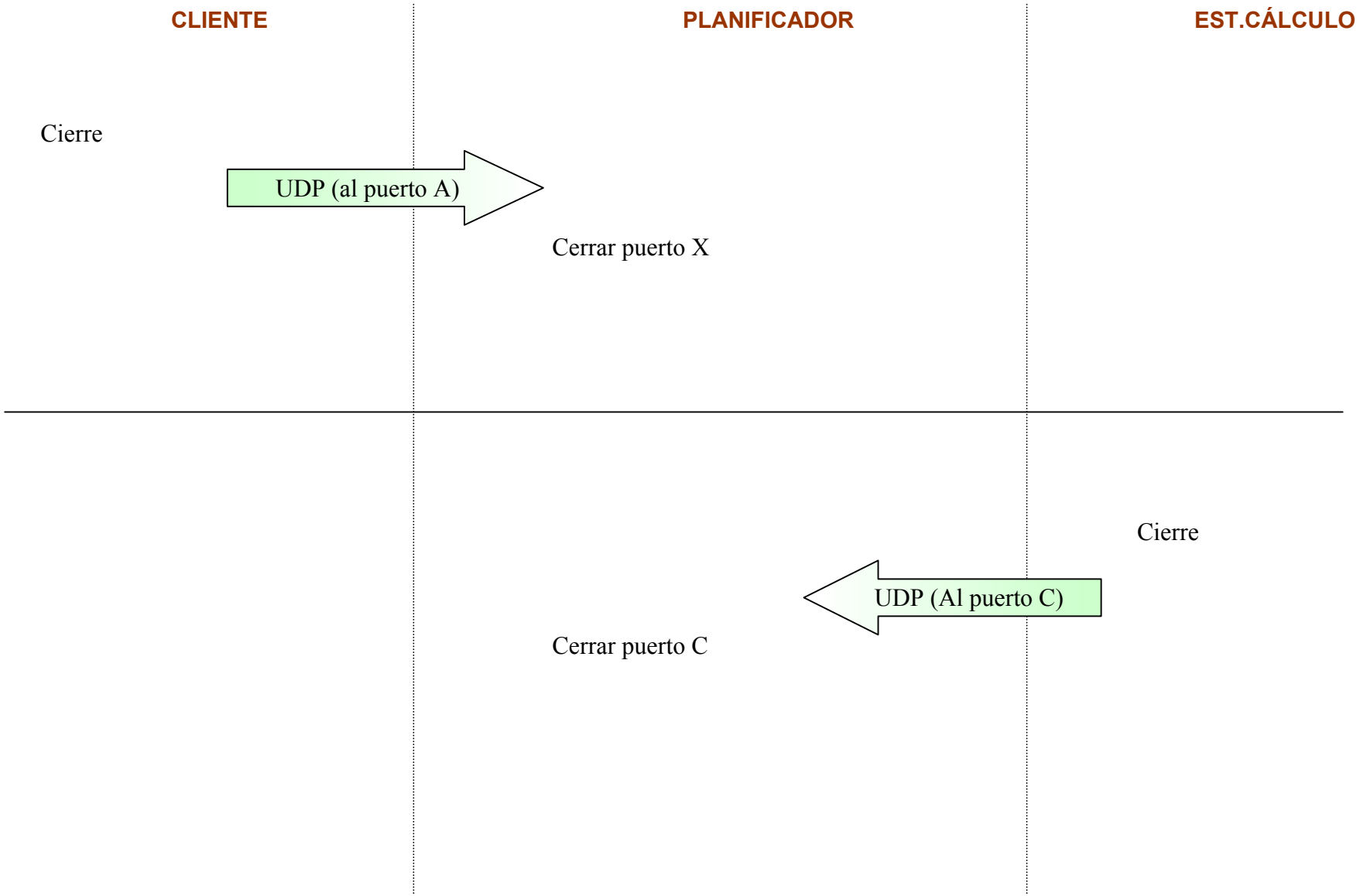
2.1.3.- Esquema de comunicaciones:Funcionamiento.



2.1.1.4.- Esquema de comunicaciones: Notificaciones al Planificador

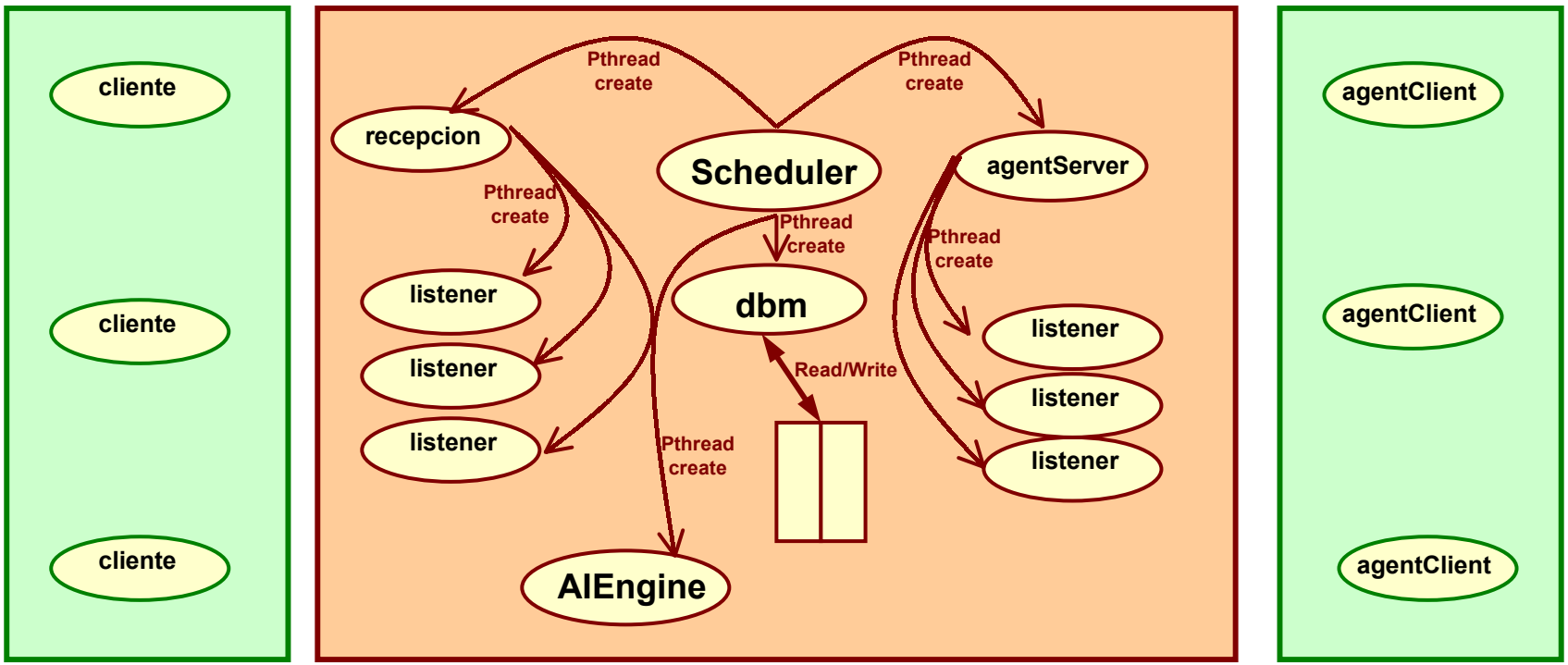


2.1.1.5.- Esquema de comunicaciones: Cierre



2.2.1.- Arquitectura Software.

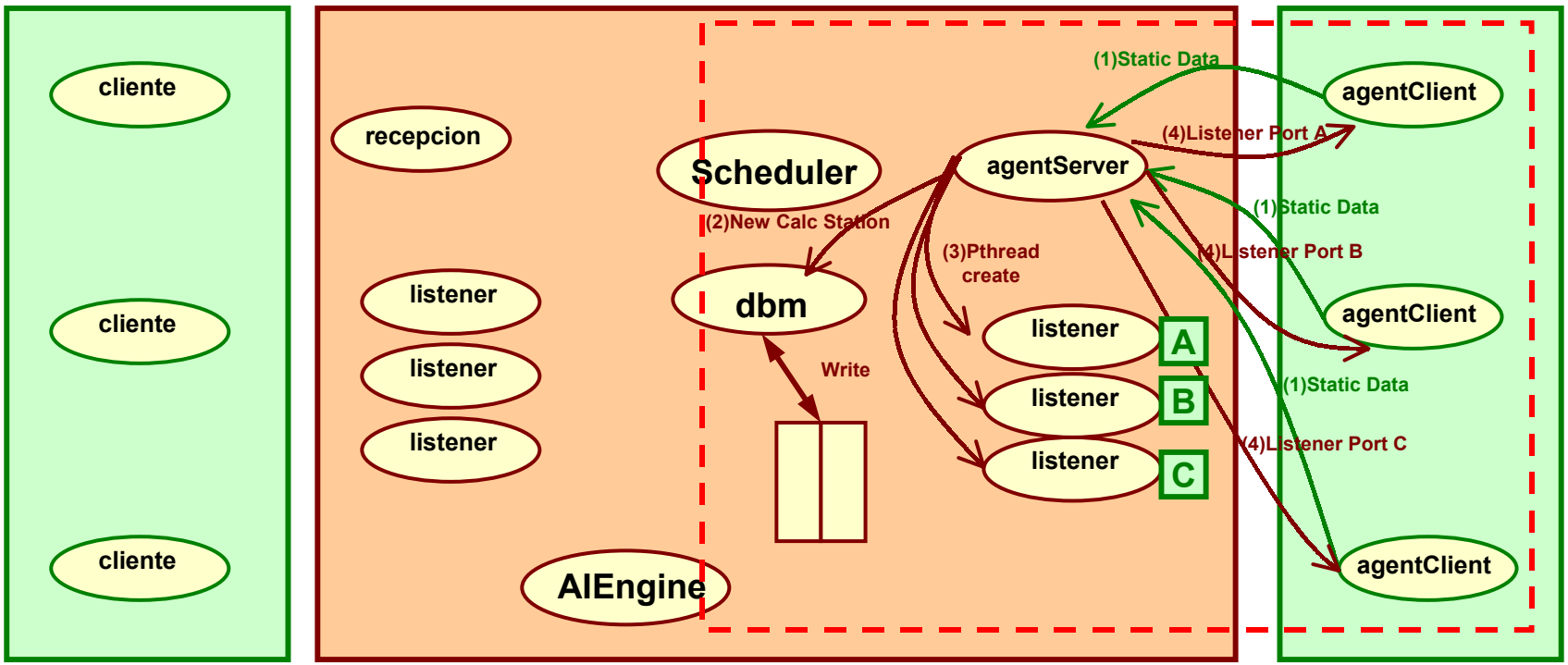
2.2.1.1.-THREADS



2.2.1.- Arquitectura Software.

2.2.1.2.- COMUNICACIONES

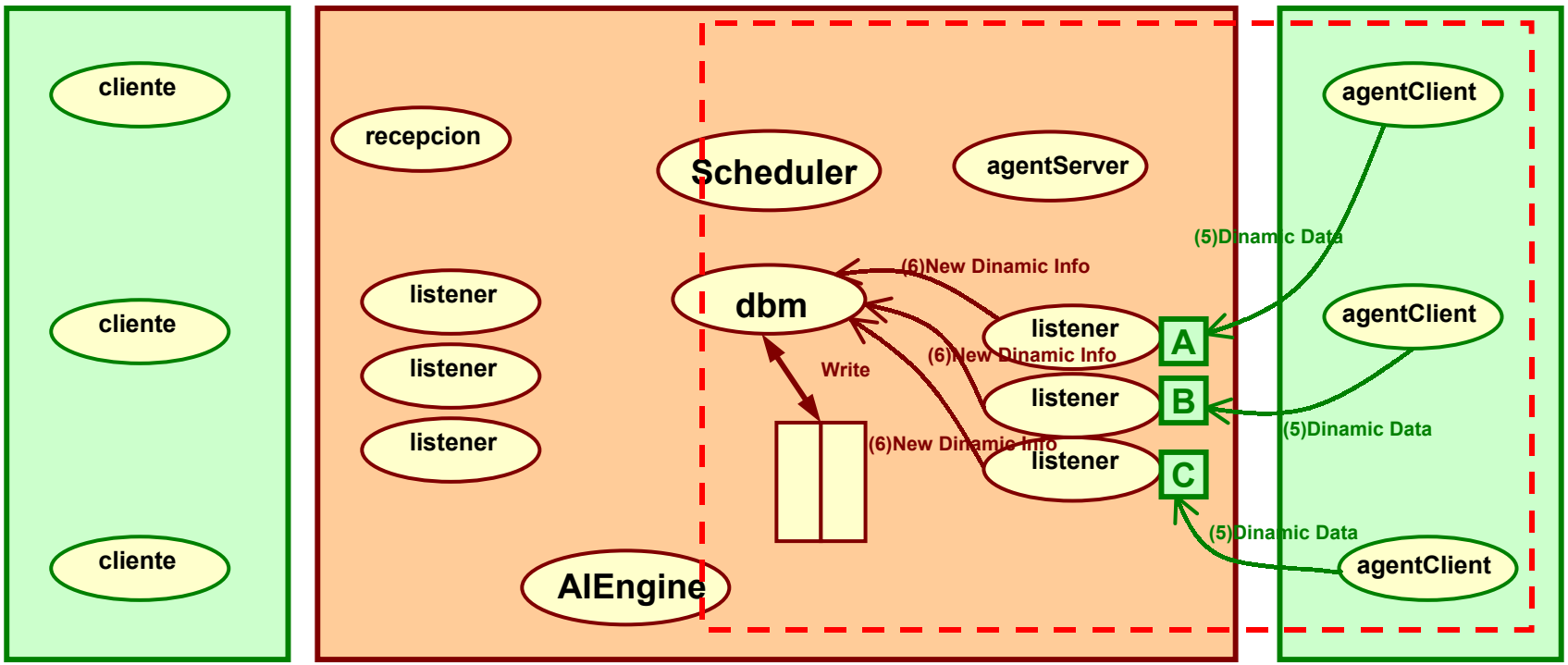
(I): Negociación de agentes



2.2.1.- Arquitectura Software.

2.2.1.3.- COMUNICACIONES

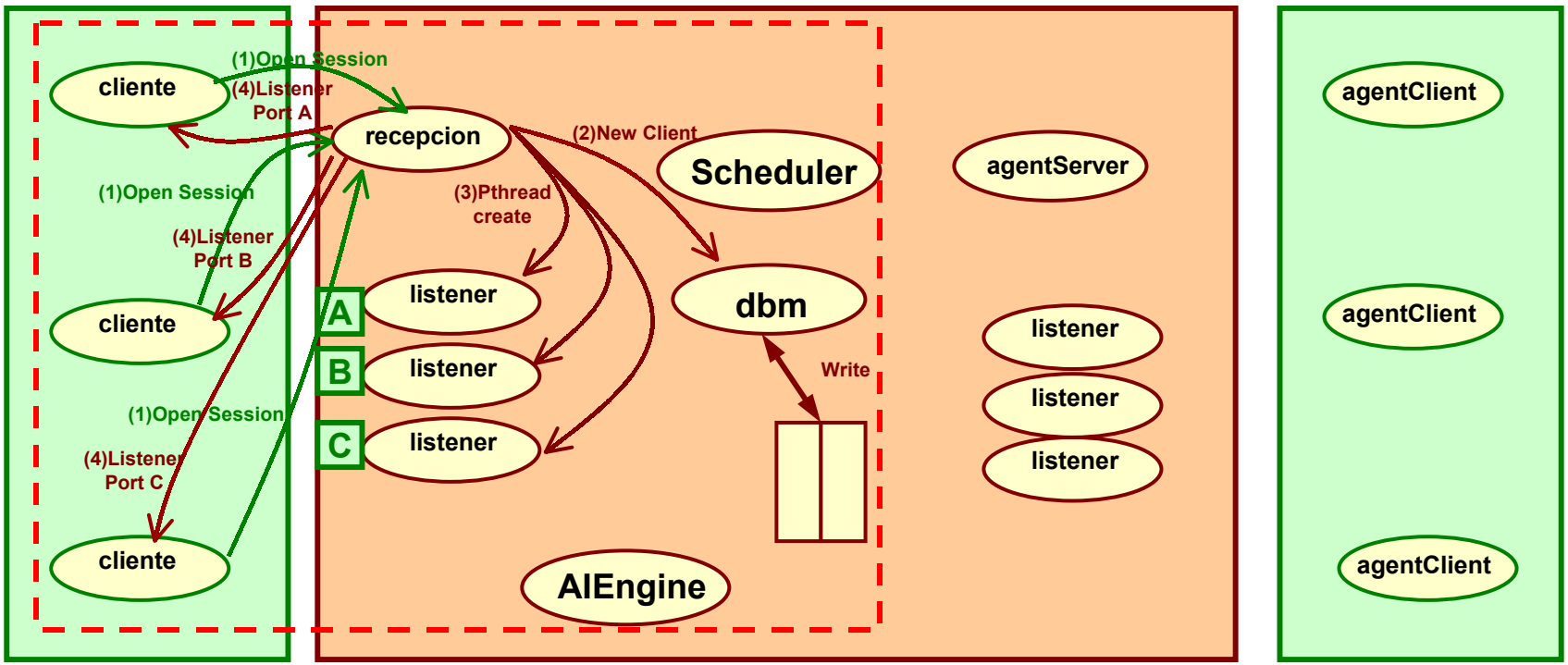
(II) Monitorización de agentes



2.2.1.- Arquitectura Software.

2.2.1.4.- COMUNICACIONES

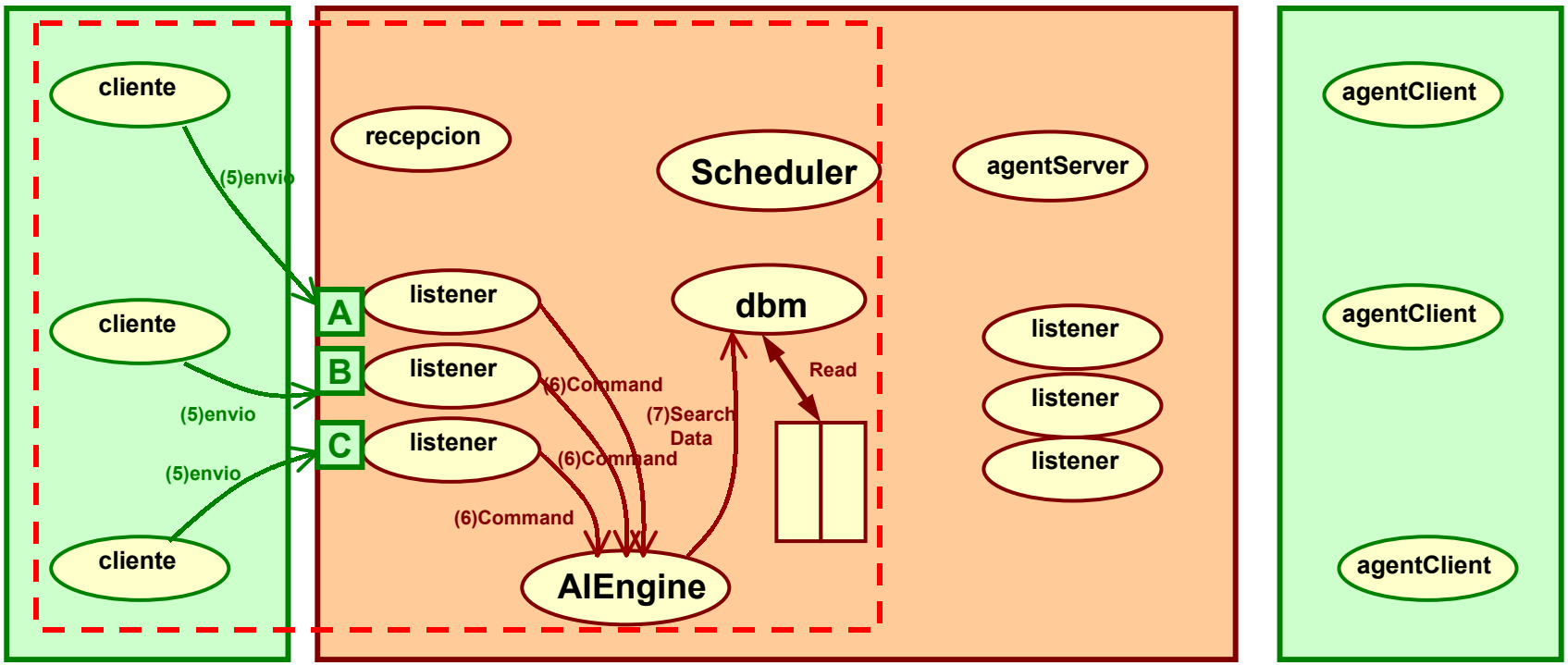
(III): Negociación de clientes



2.2.1.- Arquitectura Software.

2.2.1.5.- COMUNICACIONES

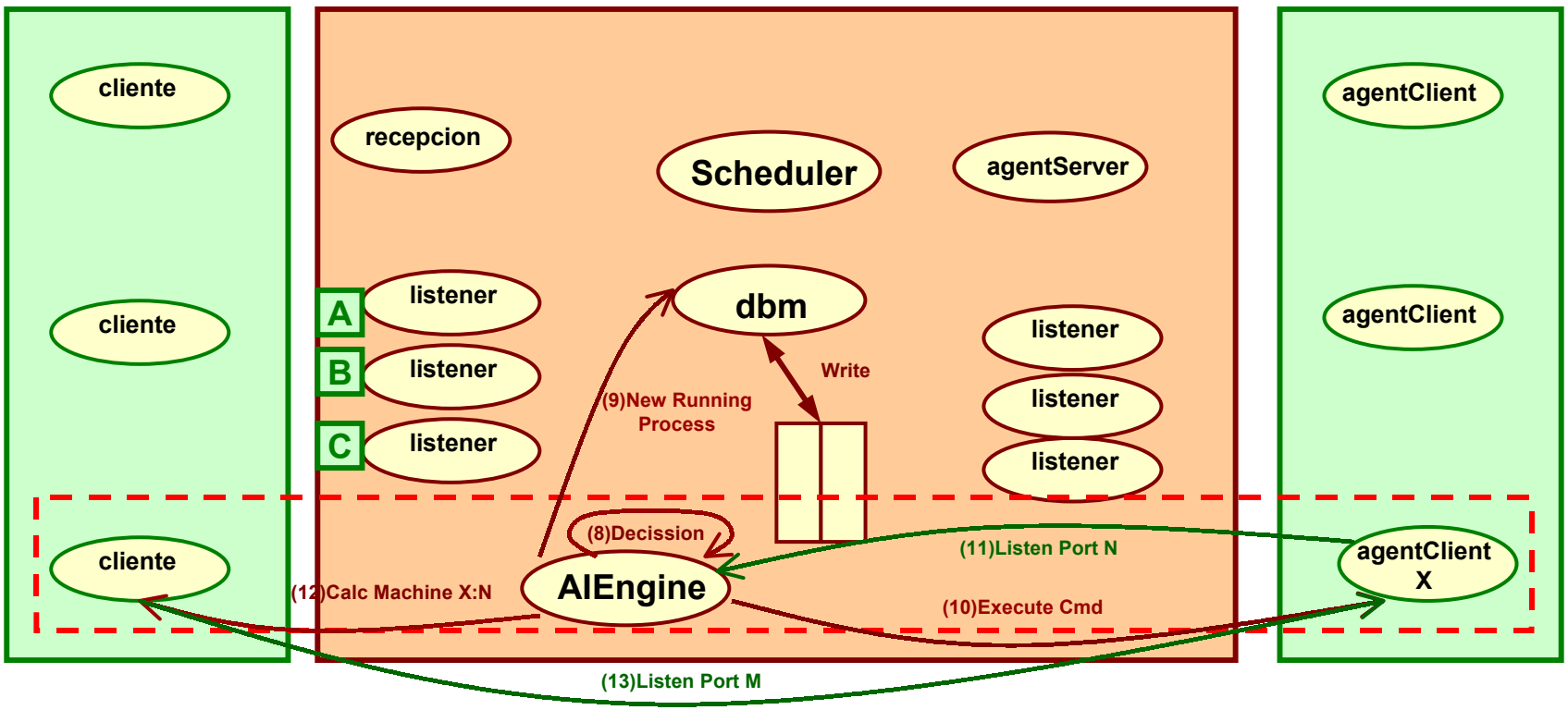
(IV): envío de órdenes



2.2.1.- Arquitectura Software.

2.2.1.6.- COMUNICACIONES

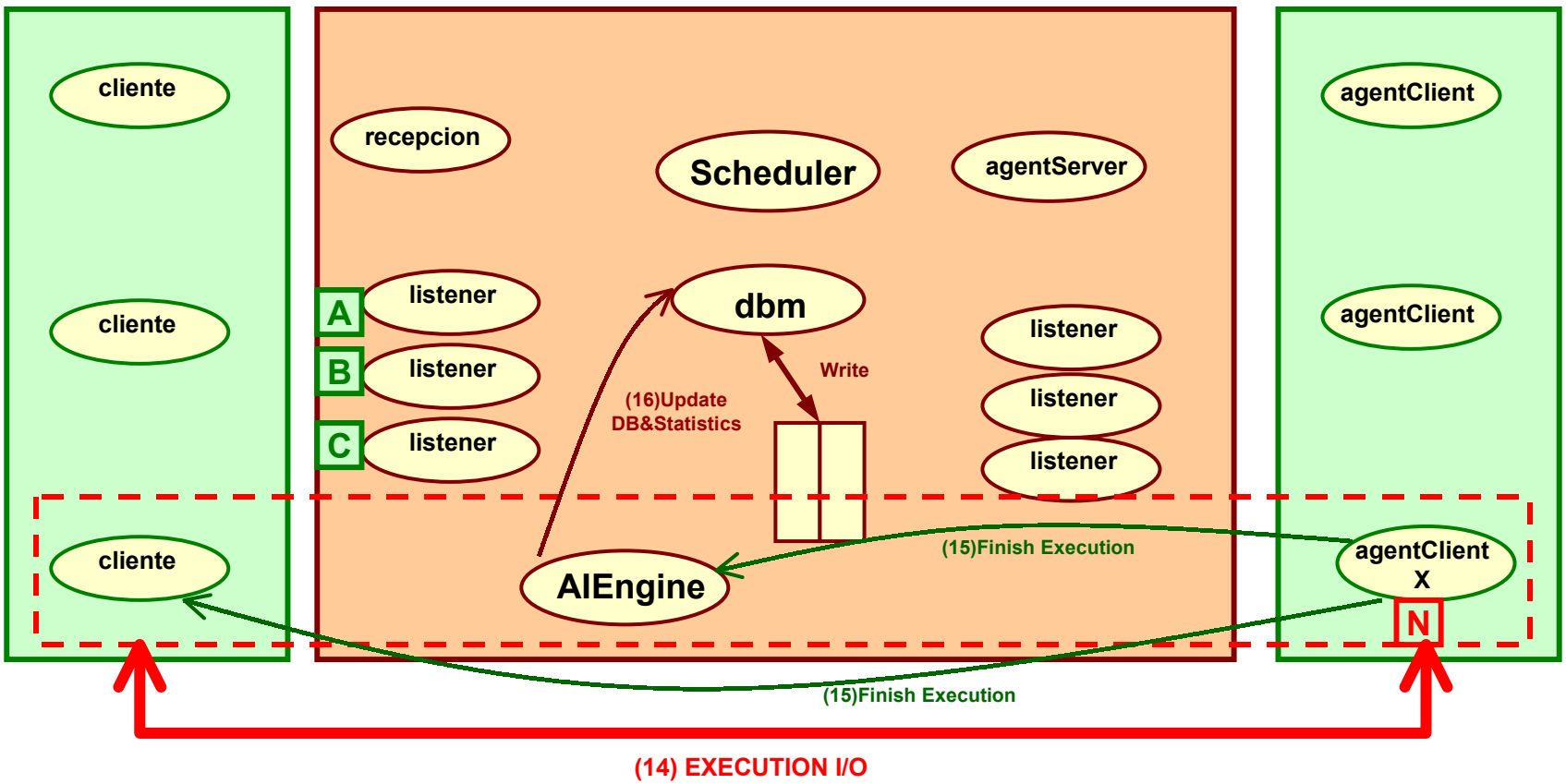
(V): Decisión



2.2.1.- Arquitectura Software.

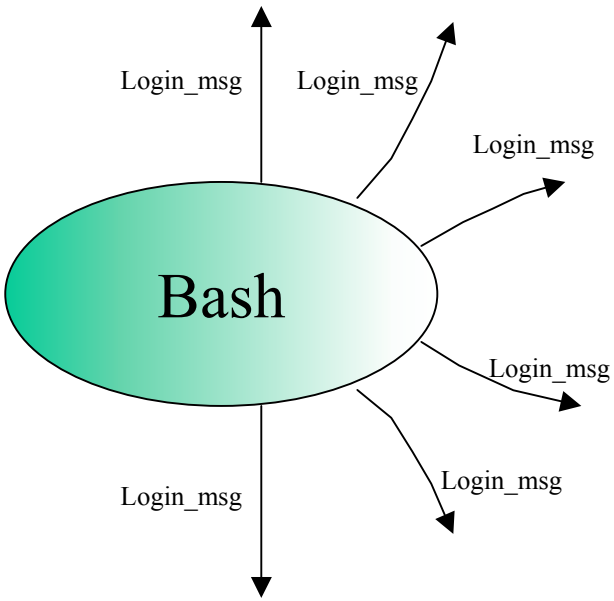
2.2.1.7.- COMUNICACIONES

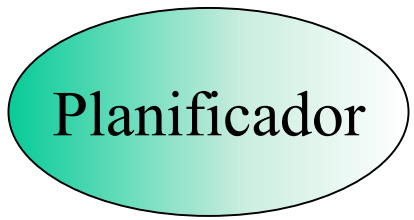
(VI): Ejecución



2.2.2.- Comportamiento global.

2.2.2.1.- Solicitud de alta.

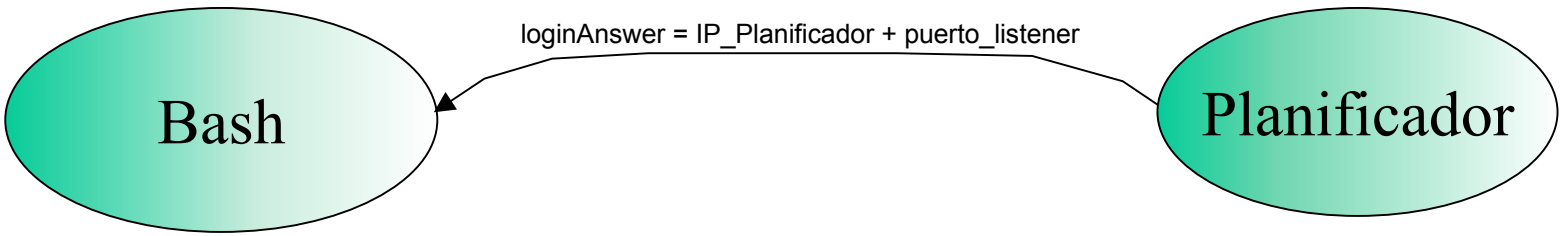




1.- Bash envía un “login_msg” en Broadcast. (CluexLogin.c).

2.2.2.- Comportamiento global.

2.2.2.1.- Solicitud de alta.

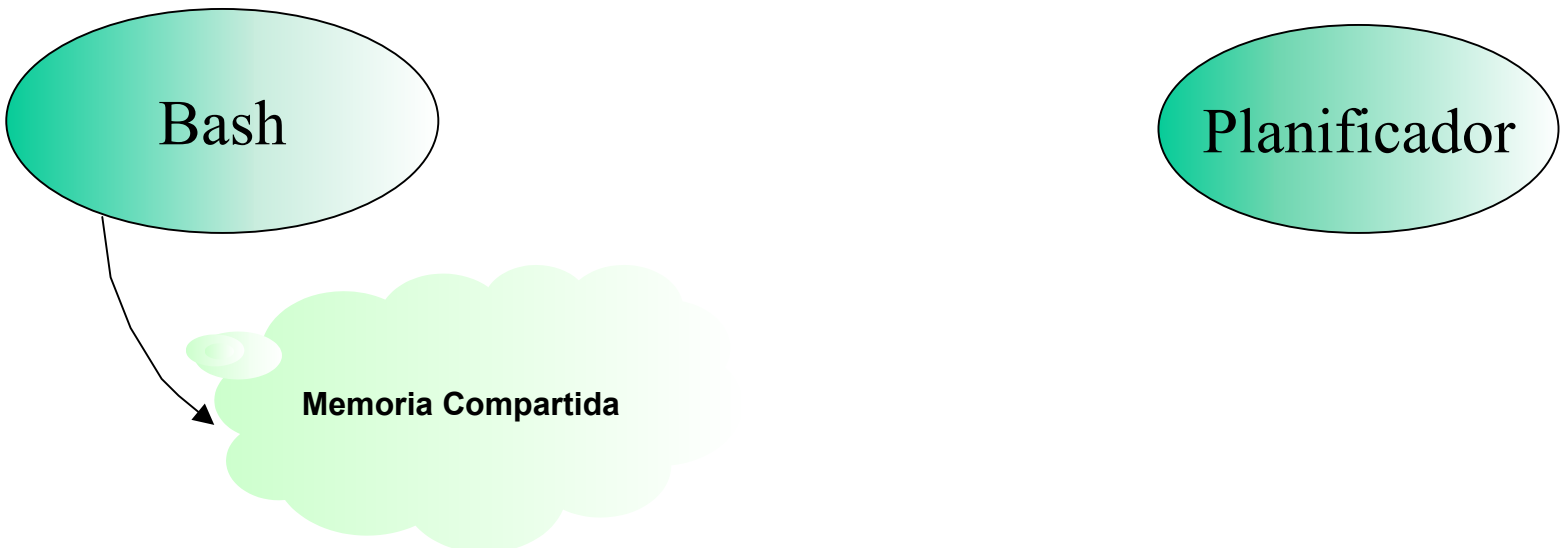


2.- Planificador recoge el “saludo” y responde, enviando al Bash su IP y el puerto de “Listener” asignado (sched_recepcion.c)

2.2.2.- Comportamiento global.

2.2.2.1.- Solicitud de alta.

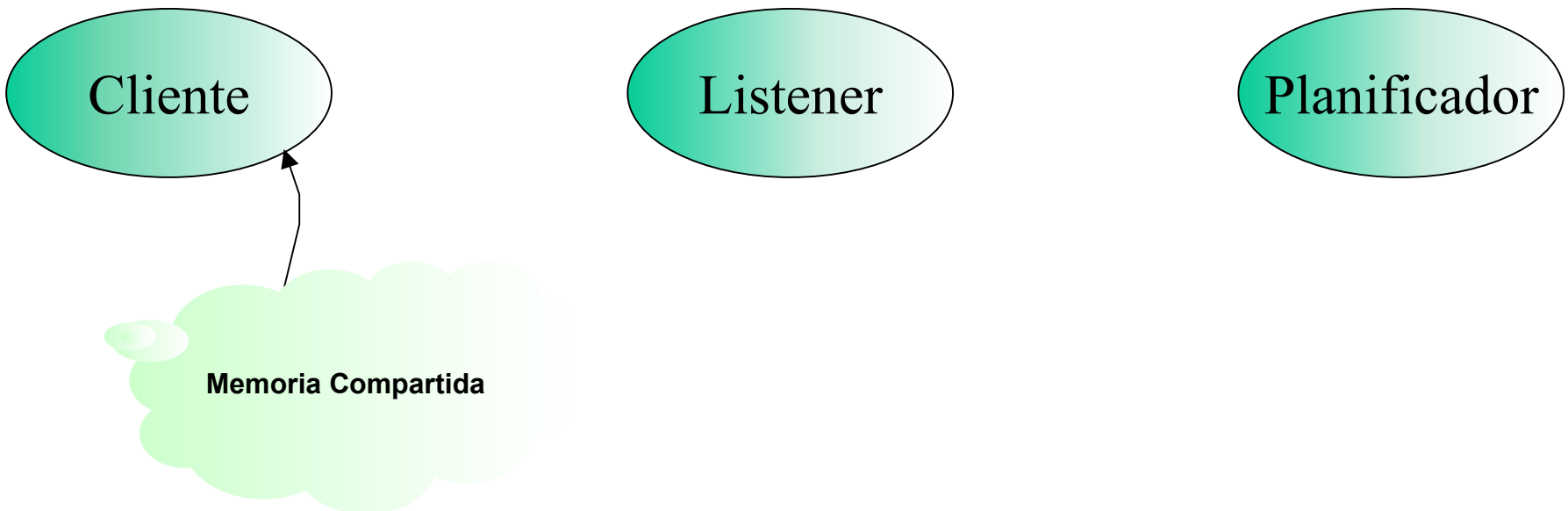
3.- Bash escribe en memoria compartida “PID_CLIENT_FILE” una línea con la siguiente información: (cluexLogin.c)



Pid_bash : puertoListener : IP planificador : clave

2.2.2.- Comportamiento global.

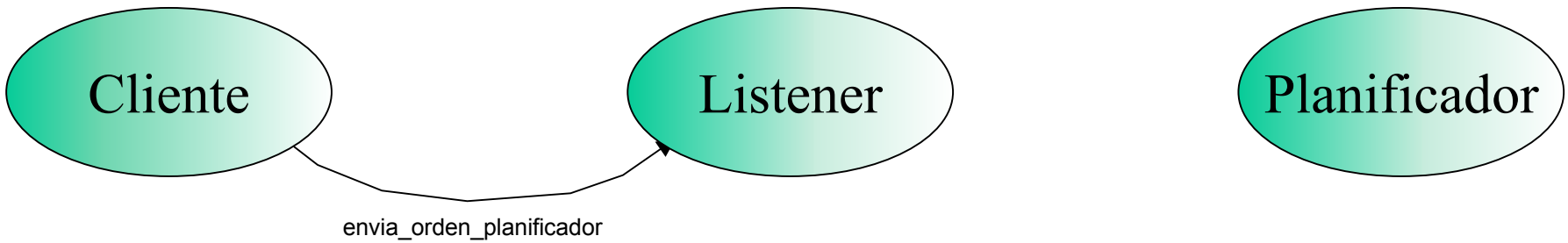
2.2.2.2.- Solicitud de tarea.



4.- Cliente lee los datos que necesita del planificador desde la memoria compartida: la IP del Planificador, y el oyente que le está escuchando.

2.2.2.- Comportamiento global.

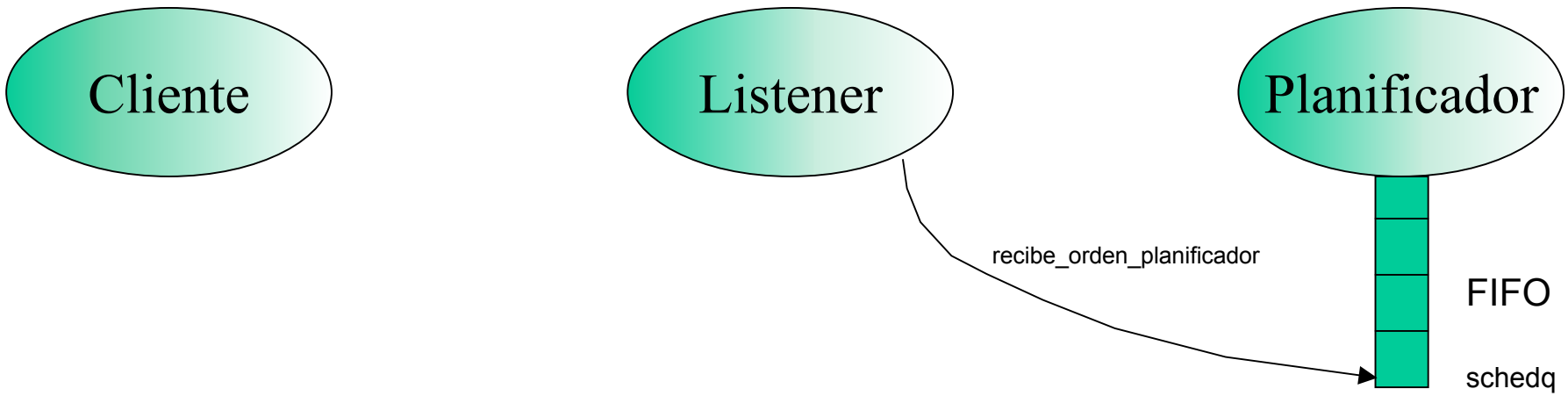
2.2.2.2.- Solicitud de tarea.



5.- Cliente envía a su Listener asignado la orden a ejecutar ("envia_orden_planificador", en cliente.c)

2.2.2.- Comportamiento global.

2.2.2.2.- Solicitud de tarea.



6.- Listener añade el trabajo recibido a la cola de trabajos pendientes del planificador (“recibe_orden_planificador”, en sched_recepcion.c)

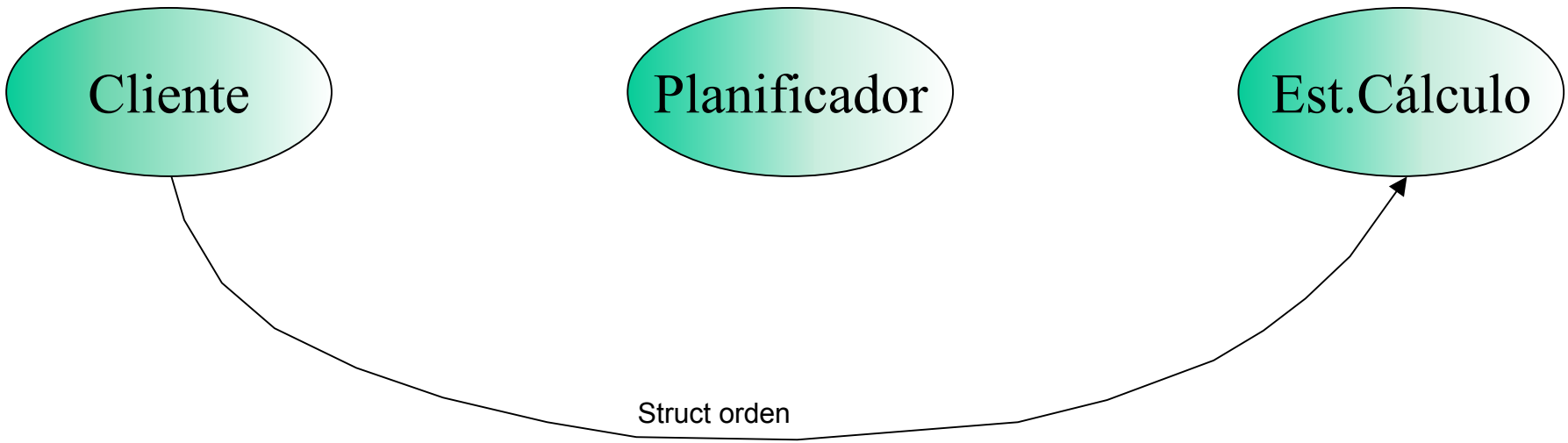
2.2.2.- Comportamiento global.

2.2.2.3.- Ejecución del trabajo.

7.-Planificador selecciona la estación de cálculo que se encargará de ejecutar la tarea y le envía la información del cliente solicitante (AIEngine.c)

2.2.2.- Comportamiento global.

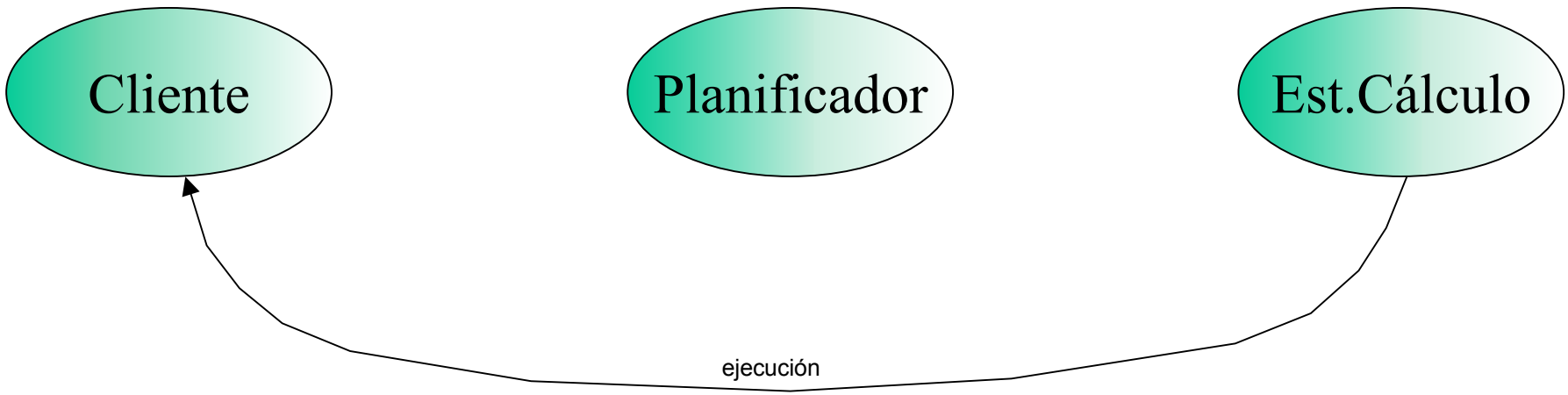
2.2.2.3.- Ejecución del trabajo.



8.- La estación de cálculo se comunica DIRECTAMENTE con el cliente para que éste le envíe la orden completa. (“ejecuta” , en Cliente.c)

2.2.2.- Comportamiento global.

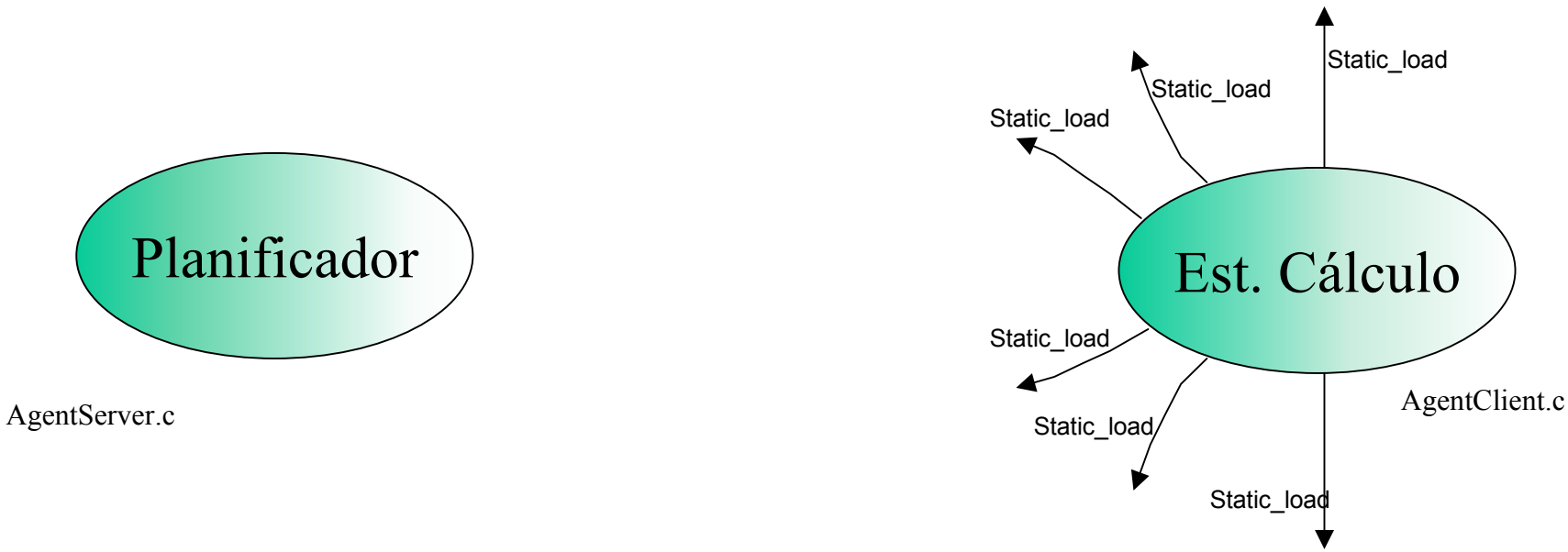
2.2.2.3.- Ejecución del trabajo.



9.- La estación de cálculo recibe la orden completa. La ejecuta, y envía al cliente la ejecución.

2.2.2.- Comportamiento global.

2.2.2.4.- Actualización de la Base de Datos.



10.- Estación de cálculo hace broadcast, enviando sus datos estáticos (no conoce la IP del planificador).

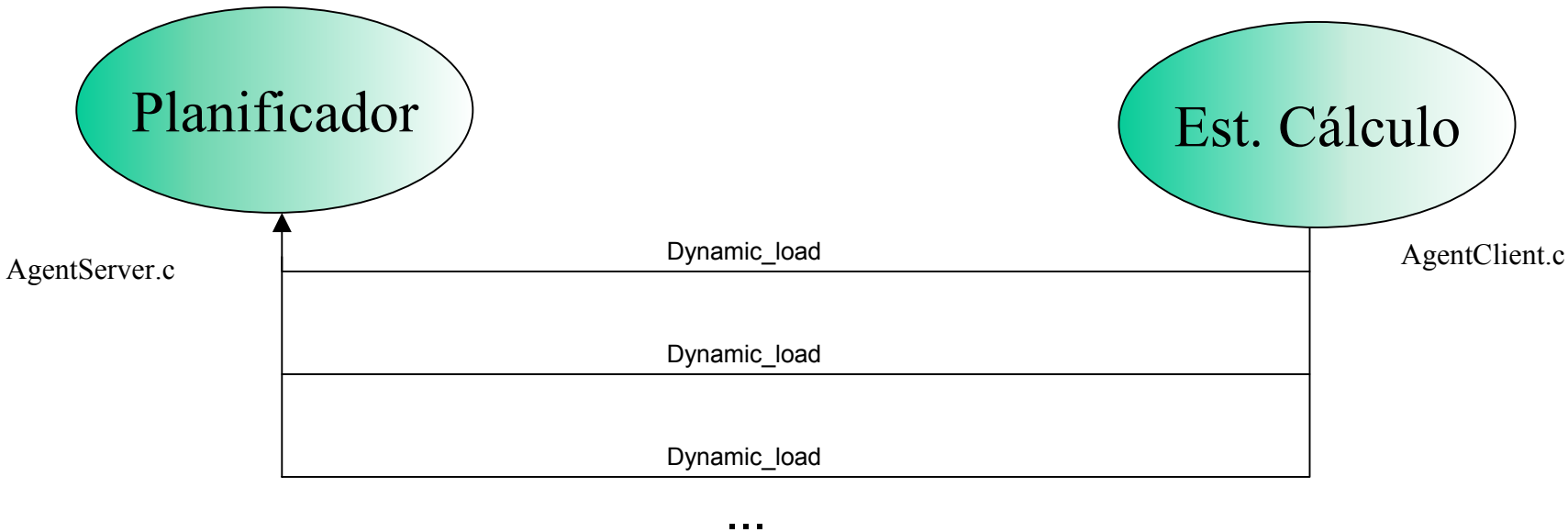
2.2.2.- Comportamiento global.

2.2.2.4.- Actualización de la Base de Datos.

11.- Planificador recoge los datos estáticos, y responde a la estación de cálculo, enviándole su IP y el puerto donde espera recibir los datos dinámicos de esa estación de cálculo.

2.2.2.- Comportamiento global.

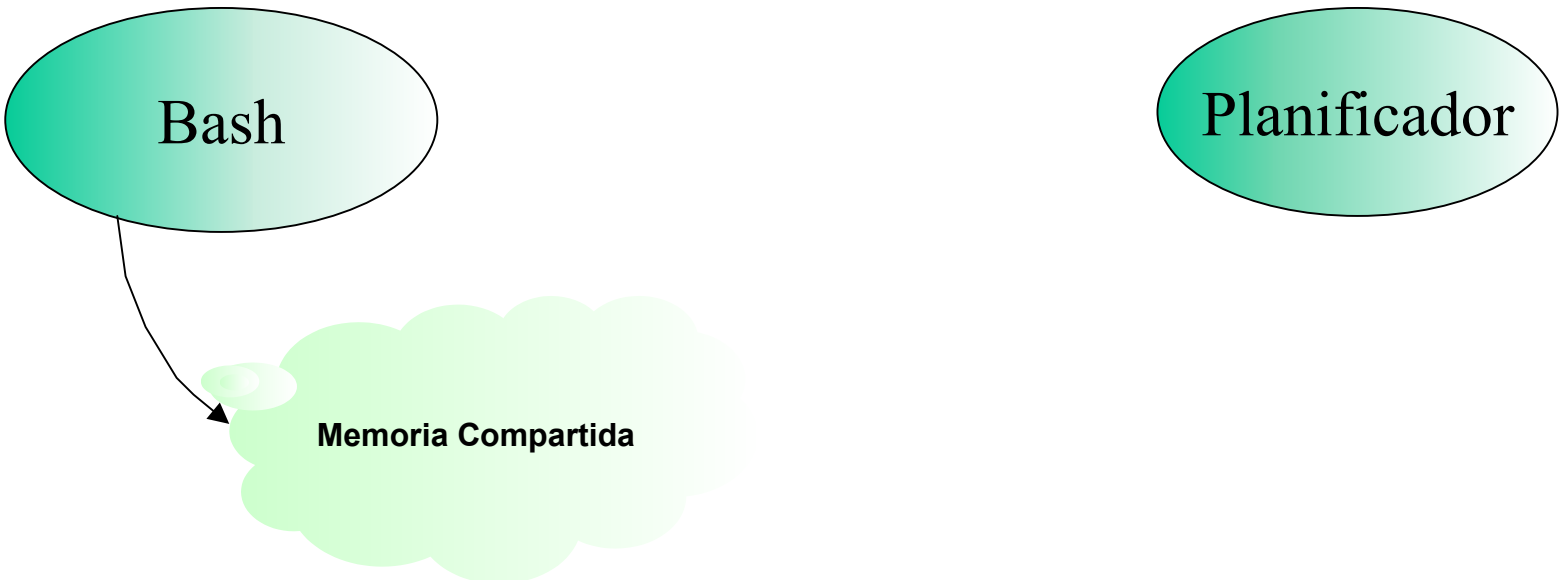
2.2.2.4.- Actualización de la Base de Datos.



3.- Periódicamente, la estación de cálculo actualizará sus datos dinámicos en la Base de Datos del Planificador.

2.2.2.- Comportamiento global.

2.2.2.5.- Solicitud de baja.

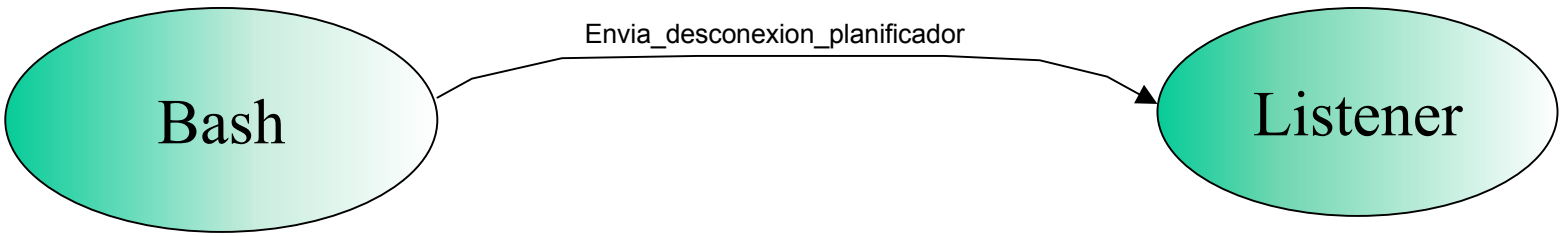


3.- Bash elimina en la memoria compartida la línea cuyo primer campo coincida con su propio pid:

Pid_bash : puertoListener : IP planificador: clave

2.2.2.- Comportamiento global.

2.2.2.5.- Solicitud de baja.



2.- Bash solicita su desconexión de ClueX al Listener del Planificador (transferencia.c)

3.- Este solicita a la BBDD del Planificador la baja de la máquina.

2.2.3.- CluexLogin.c

2.2.3.1.- Introducción.

- Se encarga de dar de alta un nuevo cliente en ClueX, y de establecer la conexión con el planificador.
  - El nuevo cliente averigua a qué planificador dirigirse cuando reciba una orden por parte del usuario.
  - Cliente almacena en un fichero la información acerca del oyente que se le ha asignado: la IP del planificador donde está, y el puerto del oyente que escuchará sus solicitudes.

2.2.3.-CluexLogin.c

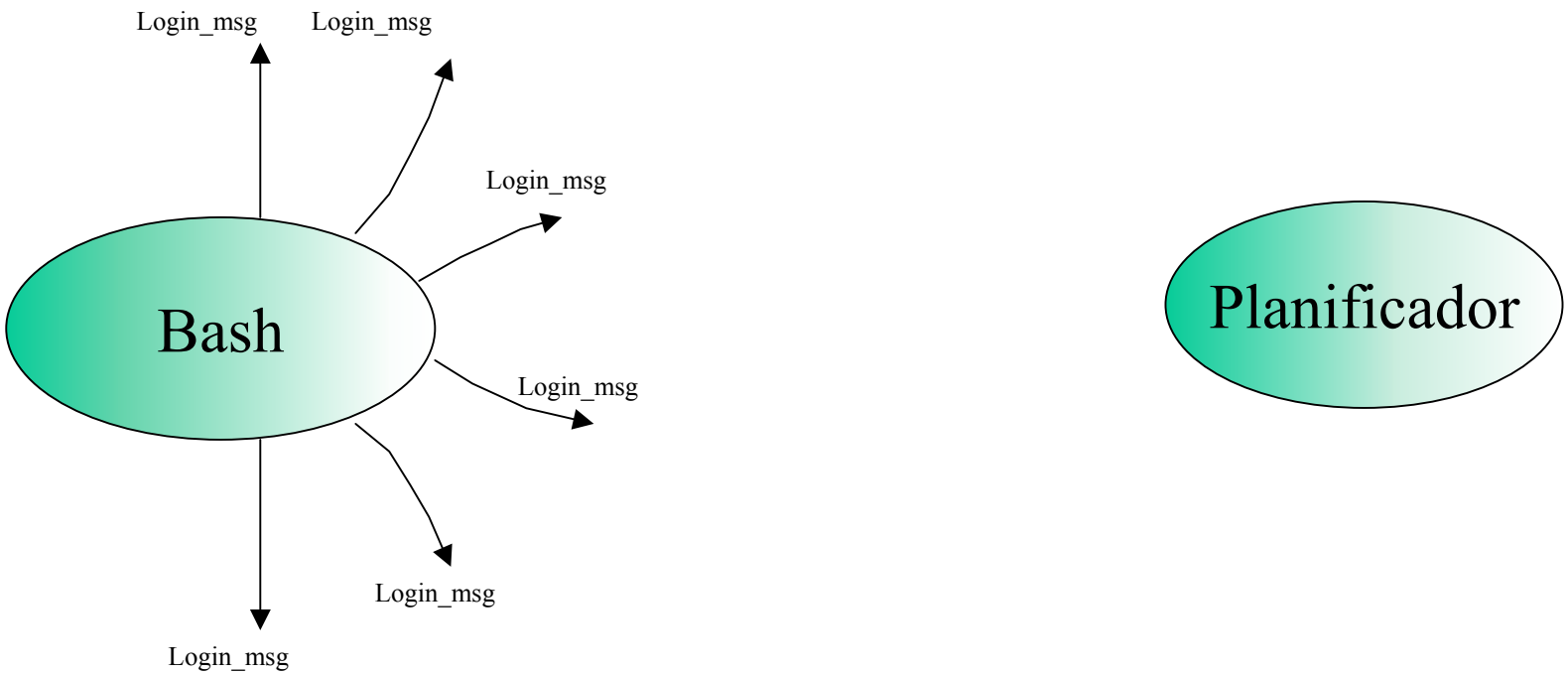
- También se encarga de dar de baja un cliente en el momento en que se desconecta de ClueX.
  - Cliente borra del fichero la entrada correspondiente a su conexión.
  - Cliente envía al Listener una solicitud de desconexión.
  - El Listener deja de escuchar.

2.2.3.-CluexLogin.c

- Cliente no conoce el planificador al que dirigirse
  - Para averiguarlo, realiza un mensaje “broadcast”, esperando que le responda un planificador.
  - Un planificador contestará al mensaje de “broadcast”, comunicando sus datos.
  - Esos datos son los que Cliente almacenará en un fichero, para ser consultado a la hora de solicitar servicio.

2.2.3.-CluexLogin.c

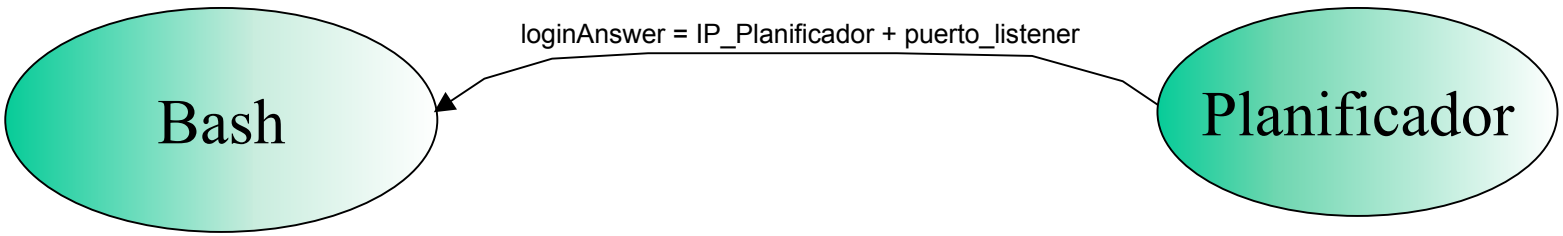
2.2.3.2.- Solicitud de alta.



1.- Bash envía un “login_msg” en Broadcast. (CluexLogin.c).

2.2.3.-CluexLogin.c

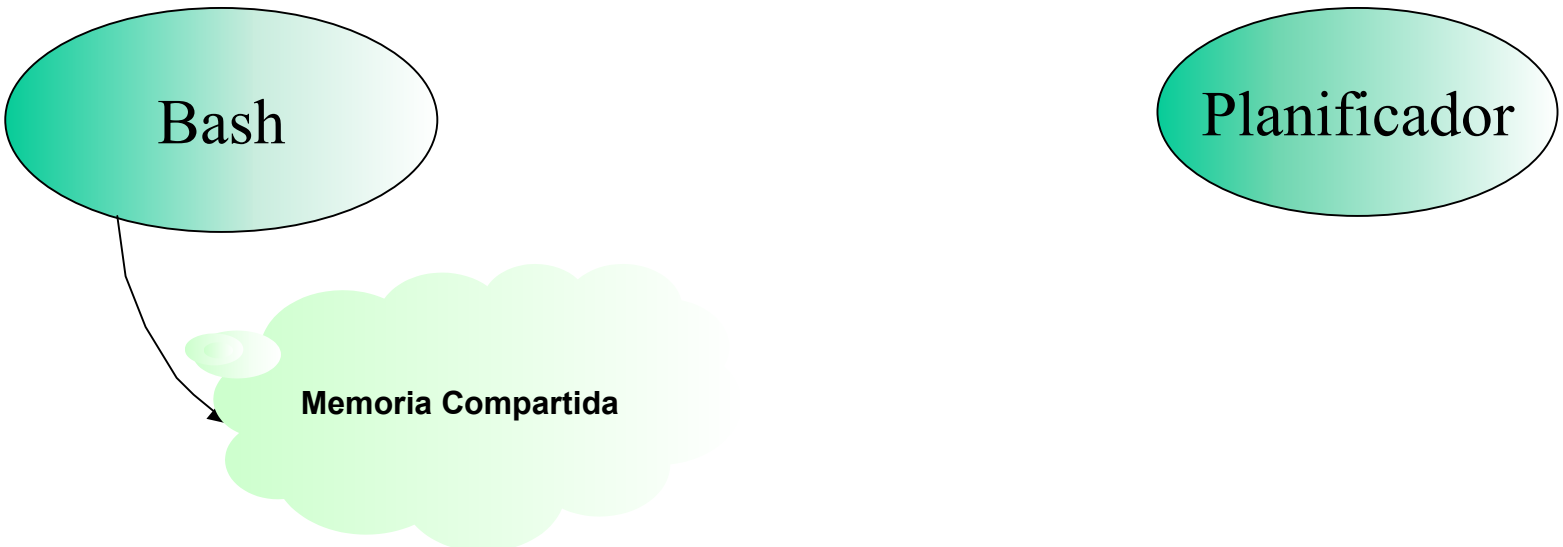
2.2.3.2.- Solicitud de alta.



2.- Planificador recoge el “saludo” y responde, enviando al Bash su IP y el puerto de “Listener” asignado (sched_recepcion.c)

2.2.3.-CluexLogin.c

2.2.3.2.- Solicitud de alta.



3.- Bash escribe en la memoria compartida una línea con la siguiente información: (cluexLogin.c)

Pid_bash : puertoListener : IP planificador: clave

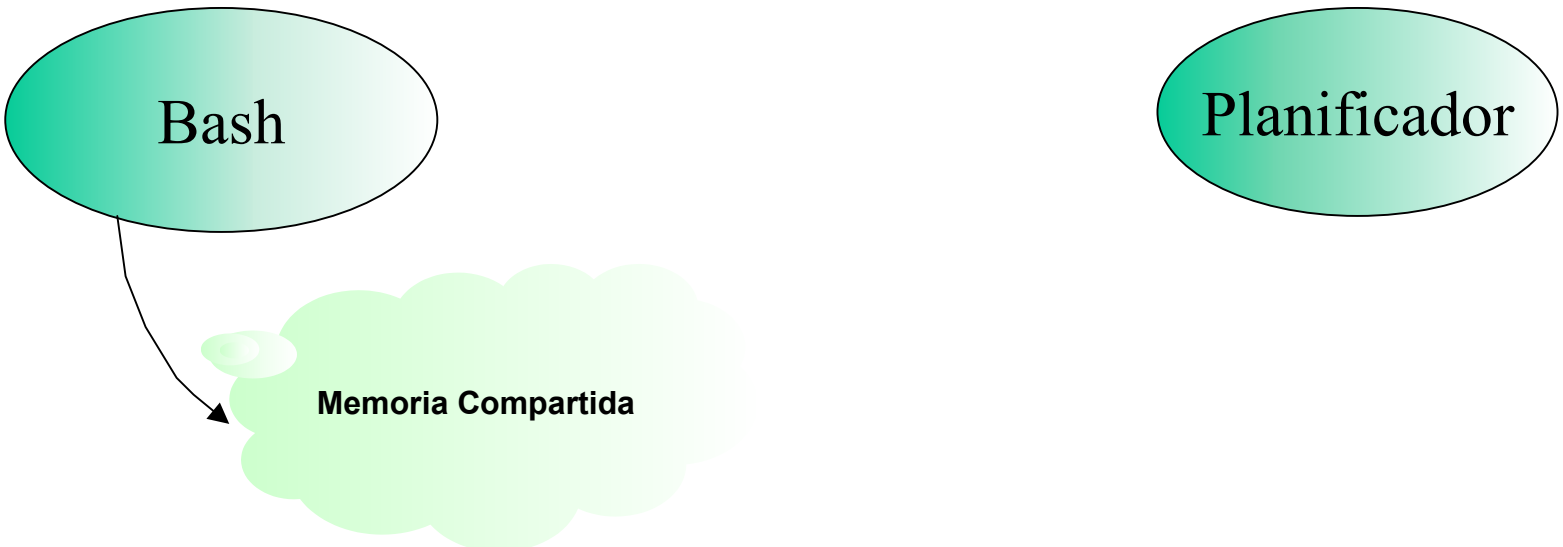
2.2.3.-CluexLogin.c

2.2.3.3.- Solicitud de baja.

- Cliente solicita su salida del sistema:
  - Busca en el fichero “PID_CLIENT_FILE” la entrada correspondiente a su pid, y la elimina.
  - Envía un mensaje de solicitud de baja al Listener que ha tenido asignado.
  - Listener da de baja la máquina de la Base de Datos del Planificador.

2.2.3.-CluexLogin.c

2.2.3.3.- Solicitud de baja.

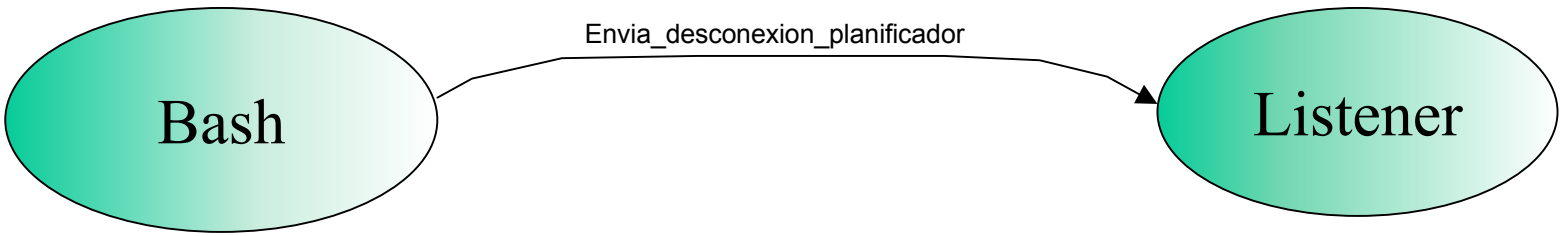


3.- Bash elimina en la memoria compartida la línea cuyo primer campo coincida con su propio pid:

Pid_bash : puertoListener : IP planificador: clave

2.2.3.-CluexLogin.c

2.2.3.3.- Solicitud de baja.



2.- Bash solicita su desconexión de ClueX al Listener del Planificador (transferencia.c)

3.- Este solicita a la BBDD del Planificador la baja de la máquina.

2.2.4.- Cliente.c

2.2.4.1.- Introducción.

- Encargado de leer los datos de entrada:
  - Obtenidos desde teclado.
  - Enviados a la estación de cálculo.
- Encargado de mostrar los datos de salida:
  - Producidos desde la estación de cálculo.
  - Mostrados por pantalla al usuario.
- Utiliza dos conexiones con la estación de cálculo:
  - Conexión de control: Información de control de la conexión, envío de señales, etc.
  - Conexión de datos: A través de este canal se envían y reciben los datos.

2.2.4.- Cliente.c

Entrada del usuario

Cliente

Conexión de Control

Conexión de Datos

Cálculo

Salida del proceso

2.2.4.- Cliente.c

2.2.4.2.- Código.

- Cliente se forma de dos procesos:
  - El primero se encarga de la lectura de teclado.
  - El segundo se encarga de mostrar los datos.
- Para saber a donde tiene que enviar las peticiones, se utiliza un archivo donde el programa cluexLogin habrá indicado el oyente correspondiente a cada sesión de usuario (identificador único).
- También se encarga de configurar la máquina para que acepte conexiones de procesos gráficos (X-Windows).

2.2.4.- Cliente.c

2.2.4.3.- Señales.

- El cliente controla las señales del sistema, reenviándolas a la estación de cálculo para que sean enviadas al verdadero proceso.
- Para ello, se utiliza el canal de control.
- Esto permite cambiar el tamaño del terminal, parar, matar, restaurar procesos...

2.2.5.- Cálculo.c

2.2.5.1.- Introducción.

- Se encarga de ejecutar el proceso en la estación de cálculo.
- Se ejecuta en forma de Demonio (cluexDaemon), recibiendo el comando a ejecutar desde el planificador y contactando con el cliente para enviar/recibir la información.
- Una vez ejecutado el proceso, envía la información de la ejecución al planificador para que éste pueda aprender de la decisión tomada.

2.2.5.- Cálculo.c

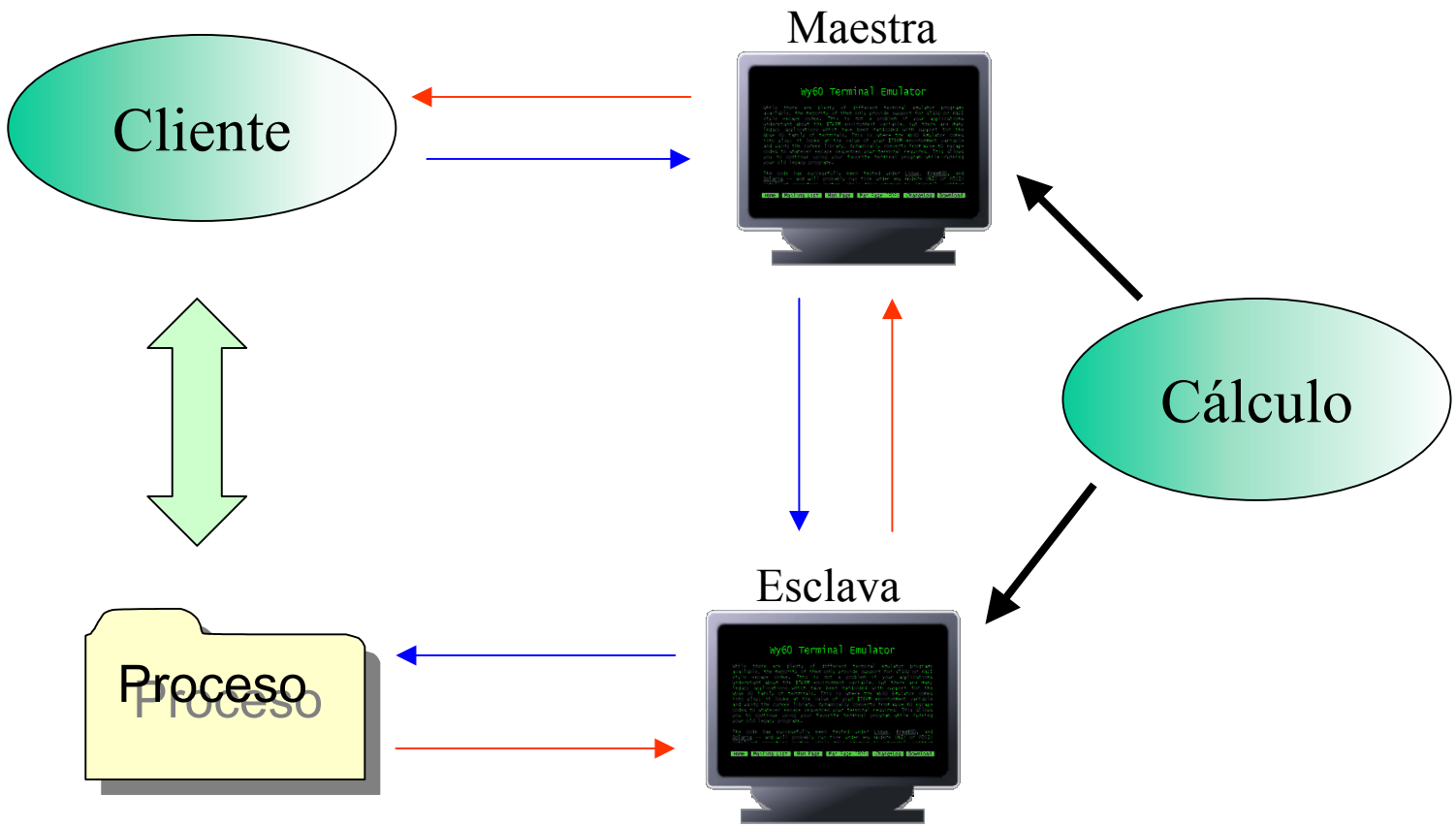
2.2.5.2.- Código.

- Para poder ejecutar el proceso y controlar su Entrada/Salida, utiliza pseudoterminales.
- La pseudoterminal se divide en maestra y esclava. Todo lo que se escriba en la esclava se leerá en la maestra, y todo lo que se escriba en la maestra aparecerá como entrada de la esclava.

2.2.5.- Cálculo.c

- Cuando se recibe una petición, el demonio se divide en 2 procesos:
  - Uno controla la terminal esclava y ejecuta el proceso solicitado.
  - El otro controla la terminal maestra, y se encarga de leer la salida generada por el proceso y enviarla a través de la conexión de datos al cliente. Análogamente, se encarga de leer los datos del usuario llegados por esa conexión y escribirlos para que la terminal esclava pueda leerlos.

2.2.5.- Cálculo.c

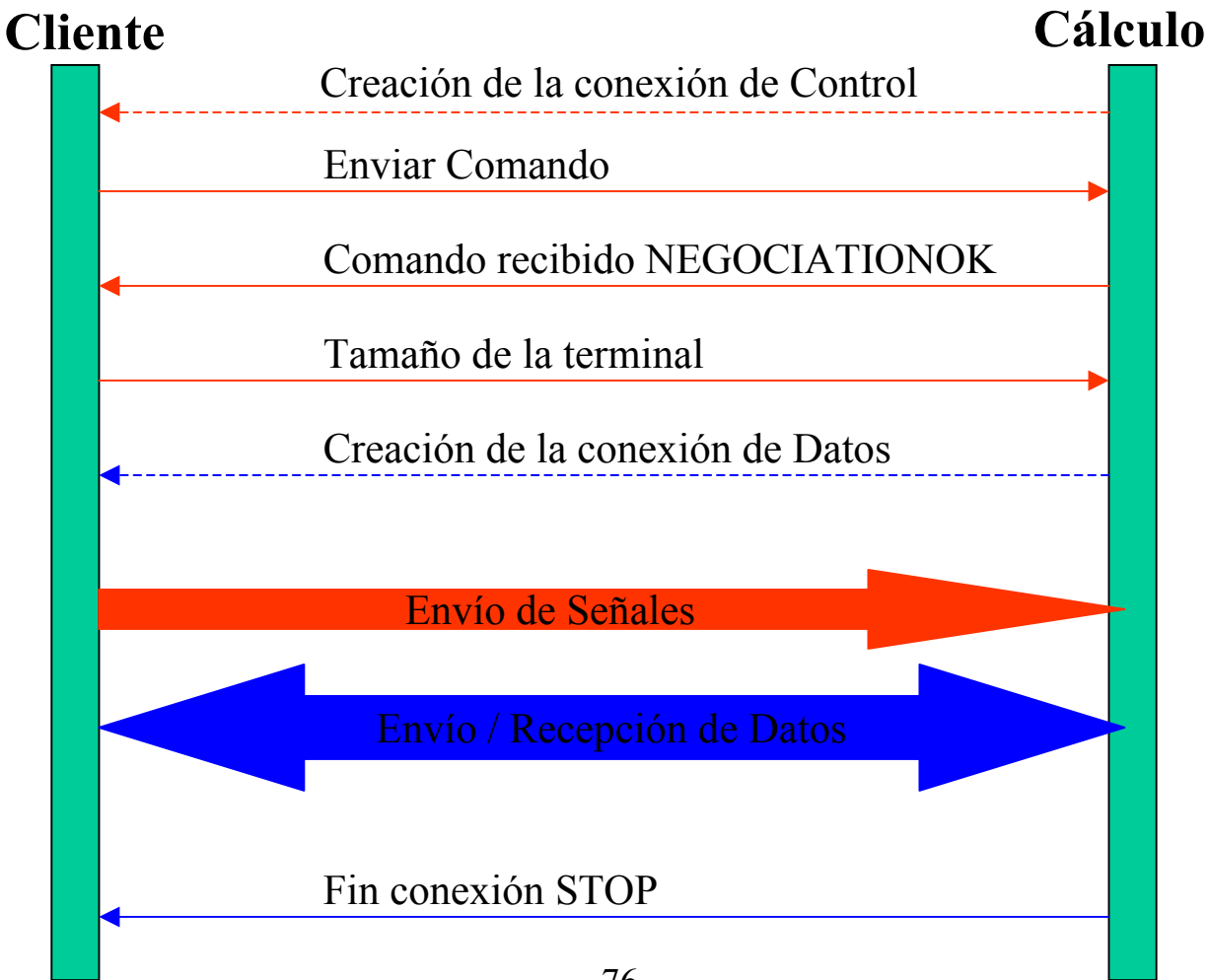


2.2.5.- Cálculo.c

2.2.5.3.- Señales.

- Análogamente a cliente.c, la estación de cálculo tiene que recibir las señales y mandárselas al proceso en ejecución.
- Como se explicó en el cliente, para manejar las señales se utiliza el canal de control de la conexión. Como el proceso encargado de enviar y recibir los datos es el que controla la terminal maestra, será éste el que lea las señales y se las envie al proceso (en la terminal esclava).

2.2.6.- Protocolo Cliente - Cálculo.



2.2.7.- Transferencia.c

2.2.7.1.- Introducción.

- Implementación de las transferencias de datos entre máquinas:
  - Cliente - Planificador
  - Cliente - Estación de cálculo.
  - Estación de cálculo - Cliente.
- Gestiona el formato de los datos a enviar, transformándolos en un flujo de bytes.
- Se encarga de transformar el flujo de bytes recibidos en el tipo original.

2.2.7.- Transferencia.c

2.2.7.2.- Transferencia Cliente - Planificador.

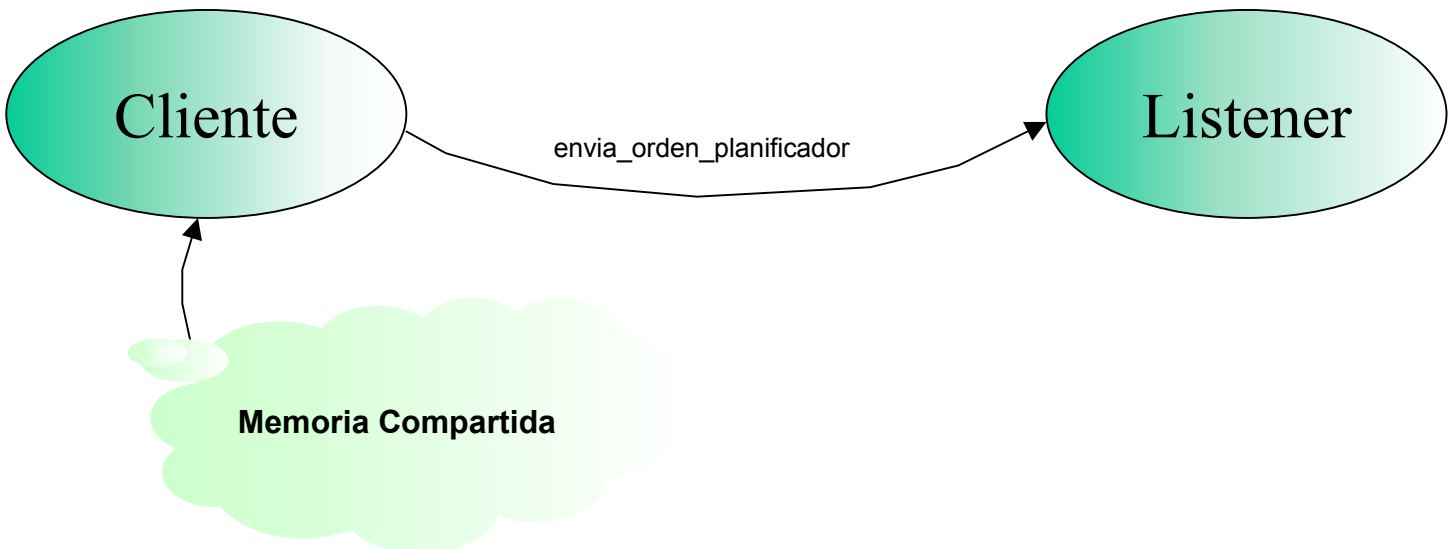
- Se ocupa de enviar el nombre de la orden a ejecutar, y los datos de la sesión cliente al "Listener" de Planificador encargado de escucharlas.
- El Listener escucha por un puerto dedicado exclusivamente para esa Sesión de cliente. (puerto UDP).
- Los datos necesarios para la transferencia son leidos por el cliente en una linea desde el fichero de configuración "PID_CLIENT_FILE".

2.2.7.- Transferencia.c

2.2.7.2.- Transferencia Cliente - Planificador.

1.- Lectura de los datos de configuración.

2.- Envío de la orden y los datos de la sesión cliente al Listener.



2.2.7.- Transferencia.c

2.2.7.3.- Transferencia Planificador – Estación de Cálculo.

- Sólo se inicia esta transferencia una vez que el cliente conoce el puerto de recepción de la ejecución de la orden.
- Es la estación de cálculo la que se pone en contacto con el cliente, informándole de su IP, mediante el establecimiento de una conexión TCP.
- Se envía la orden completa, y el puerto por donde el cliente espera la ejecución. Lo realiza la función “envia_orden_puertos”.

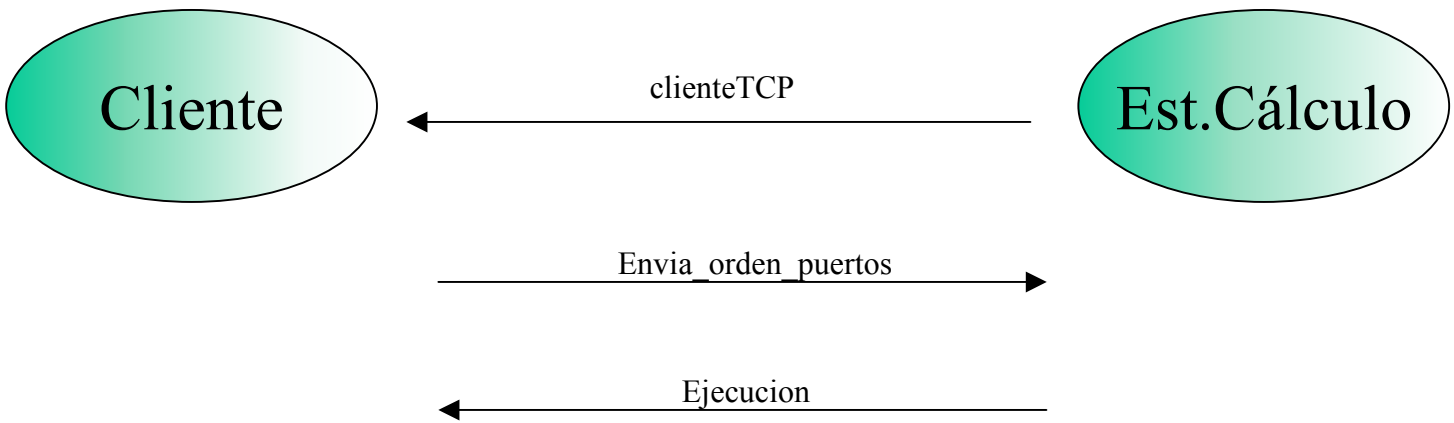
2.2.7.- Transferencia.c

2.2.7.3.- Transferencia Planificador – Estación de Cálculo.

1.- Estación de cálculo establece conexión con el cliente solicitante.

2.- Cliente manda el flujo de bytes correspondiente a la orden completa.

3.- Estación ejecuta la orden y devuelve la ejecución a Cliente.



2.2.8.- Red.c

2.2.8.1.- Introducción.

- Módulo formado por los procedimientos de envío y recepción de datos a través de conexiones TCP/IP.
  - Utilizado por todas las máquinas de ClueX.
  - Diseñado para lograr el mayor grado de encapsulación posible, ya que es el único módulo que maneja las llamadas al sistema encargadas de realizar estas comunicaciones.
  - Será el módulo a ampliar para incluir la funcionalidad de encriptación AES en el sistema.

2.2.8.- Red.c

2.2.8.2.- Comunicación TCP.

- Las funciones “clienteTCP” y “servidorTCP” son las encargadas de establecer un canal dedicado para la comunicación.
  - “clienteTCP” establece la conexión desde la máquina cliente (emisora del mensaje), abriendo el socket cliente.
  - “ServidorTCP” lo hace desde la máquina servidora (receptora del mensaje).

2.2.8.- Red.c

- “aceptarTCP” permite la recepción de datos TCP, y proporciona la IP de la máquina cliente.
- “enviarTCP” es la función de envío de datos a través de un canal TCP.
- “recibirTCP”, ejecutada por el servidor, aguarda la llegada de datos por el canal TCP.
- “recibirBloqueTCP” es una variante de la función anterior, que permanece aguardando datos del canal, hasta recibir una cantidad determinada de bytes.

2.2.8.- Red.c

2.2.8.3.- Comunicación UDP.

- No es una comunicación orientada a conexión, por lo que no hay que ligar el socket a ningún puerto, sino que son los propios paquetes los que especifican IP y puerto al que van dirigidos.
  - "servidorUDP" abre un socket desde la máquina emisora, para recibir los paquetes.

2.2.8.- Red.c

- "enviarUDP" es la función usada por el cliente para enviar datos. Se ha de especificar IP y puerto de la máquina destino.
- "enviarDifusionUDP" es una variante de la anterior, que realiza el envío "broadcast"; es decir, a todas las máquinas del sistema.
- "recibirUDP" la usa el servidor para recibir los datos.

2.2.8.- Red.c

2.2.8.4.- Otras funciones de interés.

- "espera_mensaje" permite que un servidor aguarde la llegada de un mensaje (TCP o UDP), sin tener que realizar una espera activa. Para ello, utiliza la llamada al sistema "select()".
- "hay_mensaje" verifica si existe un mensaje por un canal. Es la versión no bloqueante de la función anterior.
- "cerrar" cierra un socket previamente abierto para una comunicación.

2.2.9.- tty.c

2.2.9.1.- Introducción.

- Módulo encargado de gestionar las pseudoterminales: abrirlas, configurarlas y obtener el control sobre ellas.
- La pseudoterminal se divide en maestra y esclava. Todo lo que se escriba en la esclava se leerá en la maestra, y todo lo que se escriba en la maestra aparecerá como entrada de la esclava.
- Se compone de 3 funciones:
  - get_master_pty().
  - get_slave_pty().
  - controla_tty().

2.2.9.- tty.c

2.2.9.2.- get_master_pty().

- Abre el primer dispositivo libre de la forma:

  /dev/ptyXY

  donde:

  X Є [pqrstuvwxyzPQRST]

  Y Є [0123456789abcdef]

- Estos son las 256 pseudoterminales que nos ofrecen los sistemas Unix

2.2.9.- tty.c

2.2.9.3.- get_slave_pty().

- Abre la pseudoterminal esclava asociada al maestra.
- Si la terminal maestra abierta corresponde al dispositivo: /dev/ptyXY, la terminal esclava correspondiente será: /dev/ttyXY.
- En esta función nos encargamos de configurar el usuario propietario y los modos de uso de la terminal esclava.

2.2.9.- tty.c

2.2.9.4.- controla_tty().

- Desliga el proceso actual de su terminal anterior y lo liga a la nueva terminal esclava.
- Esta función depende mucho del sistema a utilizar ya que las llamadas que utiliza están muy relacionadas con la implementación y familia del sistema Unix utilizada.
- Por defecto está configurada para Linux, aunque también se incluyen formas alternativas para Solaris y otros S.O. Unix System V y BSD.

2.2.9.- tty.c

- Los pasos seguidos en esta función son:
  - `setsid()`:

    El proceso que realiza la llamada se convierte en líder de la nueva sesión y líder de grupo de procesos del nuevo grupo de procesos. Esta llamada cambia de un S.O. a otro.
  - `ioctl(fd, TIOCNOTTY, NULL)`

    Desconecta de la terminal actual.
  - `ioctl(ttyfd, TIOCSCTTY, NULL)`

    Controla la terminal esclava.
  - `dup2(...)`

    Redirige los descriptores de entrada/salida del proceso actual a la terminal esclava.

2.2.10.- Agentes de recolección de datos.

2.2.10.1.- Cluex *AgentServer*.

- Centraliza la recolección de datos de las estaciones de cálculo de ClueX.

- Se encarga de recibir la entrada de nuevas máquinas en el cluster y asignarles un listener dedicado.

- Envía los datos estáticos de todas las estaciones al dbm.

2.2.10.- Agentes de recolección de datos.

2.2.10.2.- Cluex *AgentClient*.

- Se inicia en cada estación de cálculo al arrancar ClueX.

- Tiene dos fases:

  • Enviar al planificador en el puerto de agentServer la información estática (struct sysinfo).

  • Enviar cada 30 segundos las medias de ocupación de memoria y CPU (struct dinamicload).

2.2.10.- Agentes de recolección de datos.

2.2.10.3.- Módulo *datacollect.c*

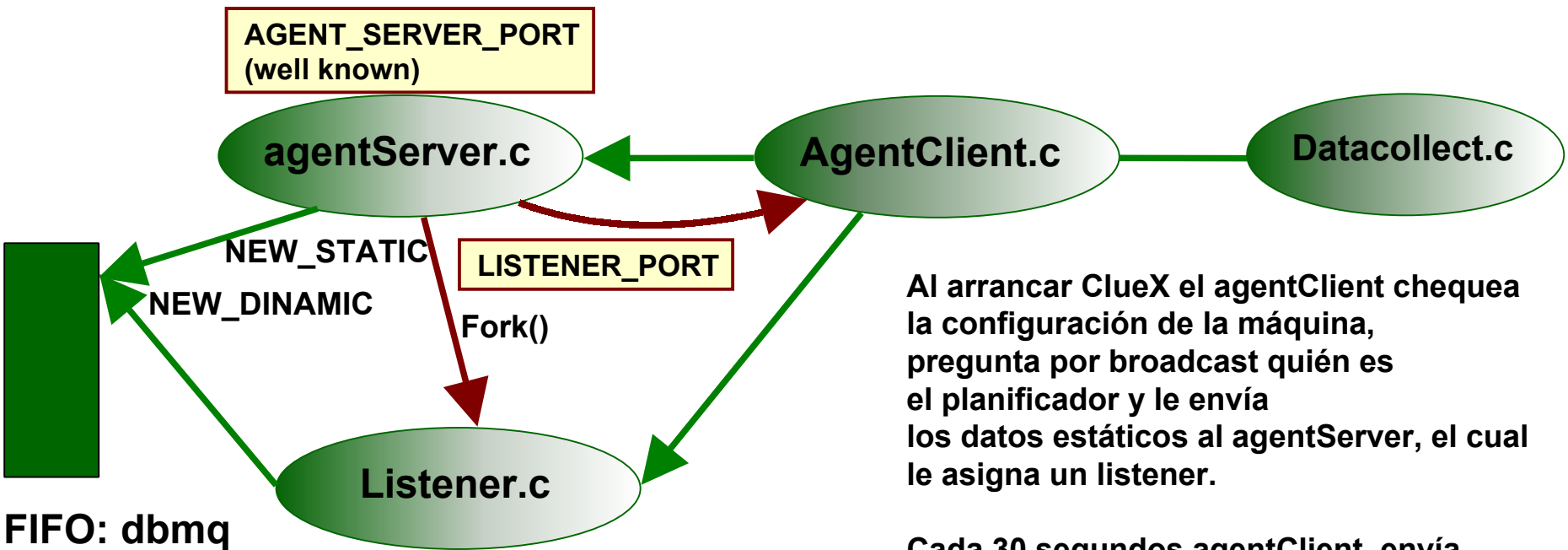
- Es la librería que implementa las funciones de medidas de rendimiento estático y dinámico en una estación de cálculo.

- Lo usa *AgentClient* para posteriormente enviarlo a *Agent Server*.

- Las funciones están diseñadas para su uso en modo *multithreading*.

2.2.10.- Agentes de recolección de datos.

2.2.10.4.- Estructura.



Al arrancar ClueX el agentClient chequea la configuración de la máquina, pregunta por broadcast quién es el planificador y le envía los datos estáticos al agentServer, el cual le asigna un listener.

Cada 30 segundos agentClient envía los datos struct dinamicload a su listener correspondiente, que se comunica con el DBM por cola de mensajes.

MÓDULO III

LA BASE DE DATOS

DE CLUEX

3.1.- La Base de Datos de Cluex: Introducción.

3.1.- Introducción.

La Base de datos del planificador debe dividirse según **funcionalidad** y **objeto**, es decir, por un lado la funcionalidad del estado de funcionamiento de la máquina (para los agentes software del SO) y, por otro, la funcionalidad de planificación de balanceo de carga, y por otro lado según el objeto sea "proceso", "máquina" o el cluster como entidad de la Base de Datos.

A continuación detallamos la Base de datos separada por funcionalidad (en términos más generales), y posteriormente separdo por objetos (visión más cercana a la implementación).

3.2.- Visión de la BD por funcionalidad.

3.2.1 Agentes software.

3.2.1.1.- Conceptos generales.

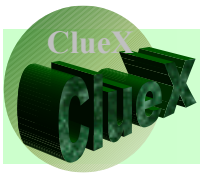
Recordemos la definición de "**agente**": **Sistema situado por encima de un cierto entorno y que tiene capacidad de actuar autónomamente en ese entorno para satisfacer sus objetivos.**

Los agentes que deseamos desarrollar tendrán los siguientes objetivos:

- **Identificar y almacenar en la Base de Datos los datos relevantes** para la evaluación de procesos con la intención de hacer una planificación eficaz.

- **Monitorizar el sistema** en todo momento; detectar problemas o conflictos que pudieran surgir, y aislar las máquinas afectadas del resto del sistema para que no influyan negativamente, similarmente a lo que realiza SNMP (Simple Network Management Protocol).

Intentaremos que los agentes desarrollados sean lo suficientemente **reactivos** como para que la información de la Base de Datos sea reciente, y los problemas en el sistema sean detectados con la mayor prontitud.



3.2.1.1.- Agentes planificadores

En principio, la base de datos clasifica los algoritmos en **clases** (matemáticos, de E/S, etc.).

Los agentes que recopilen datos para lograr una mejor planificación deberán tener en cuenta **los siguientes parámetros por cada clase de proceso**:

- Relación carga CPU / carga E/S.
- Duración (podemos tomar la duración media, o asignar mayor peso a los que se ejecutaron más recientemente, como en ciertos algoritmos de planificación de monoprocesadores).
- Sobrecarga de recursos de red que produce.
- Relaciones entre procesos (comunicaciones entre ellos, dependencias, etc.). Esto permitiría agruparlos a la hora de su ejecución, y que cuando lo hagan, se encuentren relativamente próximos, para facilitar esas relaciones.

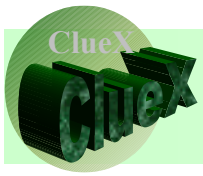
Por otro lado, **la Base de Datos deberá tener datos de las estaciones de cálculo del sistema:**

- Cuántos procesos esta ejecutando cada una (aunque este valor puede resultar engañoso, pues los demonios ejecutándose pueden falsear la apariencia de disponibilidad de las máquinas). Esta información deberá ser actualizada regularmente, ya que, además de la asignaciones que el planificador realice, éstos procesos pueden hacer llamadas “fork”, y generar procesos hijos.
- Qué porcentaje de CPU de cada estación está ocupada en el momento actual (será un dato mucho más fiable).
- Caracterísiticas físicas de las estaciones: de procesador, memoria, capacidad de comunicación, etc.
- Detección de equipos caidos de la red para no ser tenidos en cuenta a la hora de planificar

3.2.1.2.- Agentes de monitorización

Estos agentes deberán tener en cuenta los siguientes parámetros para detectar anomalías en el sistema:

- Detección de sobrecarga de tráfico de red en una parte del sistema.
- Estaciones que no responden o permanecen inactivas durante un tiempo.
- Procesos con duración excesiva.



3.2.1.3.- Otros datos relevantes

La Base de Datos deberá albergar los siguientes datos para evitar los problemas críticos mencionados en esta memoria inicial:

- **Diferenciar entre procesos de consola y procesos gráficos**, ya que el ripo de conexión entre cliente y el procesador de ejecución será distinto en uno u otro caso. Además, en el segundo caso, se debe disponer de la IP del cliente donde se deban mostrar los gráficos.
- **Detección de problemas producidos por procesos que utilicen mecanismos IPC para comunicarse**. A priori, una solución factible sería forzar la migración de uno de los procesos implicados en la comunicación al procesador donde estuviese el otro, antes de gestionar la comunicación IPC.
- Queremos que el sistema sea fácilmente escalable, y hemos contemplado la posibilidad de tener **más de un planificador**. En tal caso, es más que probable que varios algoritmos de planificación requieran que dichas máquinas planificadoras deban comunicarse para intercambiar información.

3.3.- Visión de la BD por objeto.

La visión de la BD por objeto es una vista más **a nivel de tablas** de base de datos. Aunque ahora no vamos a entrar en cómo estarán organizadas estas tablas físicamente, podemos dividir el tipo de objetos de información en ClueX en cuatro partes: Usuarios, Procesos, Máquinas y ClueX. Además también hay que mantener una tabla con procesos lanzados actualmente, que será información dinámica que luego se volcará en el resto de tablas para actualizar la información.

3.3.1.- Información de Usuarios.

Respecto a los usuarios, queremos crear el perfil de ejecución típico de un usuario en concreto, con lo que lo que queremos es tener una lista de los procesos que un usuario arranca al menos el 25% de las veces que hace login. Esto es una medida relativa, también podemos usar una medida absoluta del número de veces que arrancó tal proceso o el número de veces que arranca en una sola sesión cada proceso.

- Username
- Lista de procesos:

    - Nombre de proceso.
    - Veces ejecutado (total).
    - Media veces ejecutado por sesión.

- Tasa de ejecución (% de veces que el usuario lo ejecuta en cada login).

3.3.2.- Información de Procesos.

De cada proceso gestionado por ClueX, es decir, cada proceso que puede ejecutar un cliente de ClueX, debemos tener la siguiente información:

- Tamaño del binario.
- Tamaño de los datos sobre los que trabaja.
- Tamaño medio de los datos sobre los que trabaja.
- Tiempo óptimo de ejecución (parámetro histórico).
- Tiempo medio de ejecución (parámetro histórico).
- Porción I/O.
- Procesos relacionados (histórico): Qué procesos se ejecutan normalmente después de ejecutar éste.
- Propósito: Ocio, profesional, etc…
- Tipo: Cálculo, web, etc…
- Localización y parámetros de ejecución.
- Proceso de texto o proceso gráfico.
- Ancho de banda de red utilizado (histórico).
- Hora del día a la que se suele ejecutar.
- Días de la semana cuando se suele ejecutar.
- ¿Mayor carga en días festivos?

3.3.3.- Máquinas

En cuanto a las máquinas queremos una información suficiente para poder dar un valor numérico de calidad de máquina, que será el que se use finalmente a la hora de planificar. Para ello a cada una de estas características se le dará un **peso** y un **baremo de puntuación**, que se sumará y tendremos como resultado esa calidad deseada. También existe la posibilidad de hacer dos medidas: calidad de proceso y calidad de I/O.

3.3.3.1.- Parámetros estáticos.

- Tipo CPU.
- Reloj de CPU.
- Reloj de placa.
- Tamaño de la RAM.
- Tamaño de la cache de 1º nivel.
- Tamaño de la cache de 2º nivel.
- MIPS pico.
- MFLOPS pico.
- Velocidad de la RAM.
- Tamaño del swap.

- Velocidad del disco (para tener una idea de la velocidad de swap).
- Tamaño del disco.
- Tarjeta gráfica.
- RAM tarjeta gráfica.
- Reloj Tarjeta gráfica.
- Velocidad del bus de la tarjeta gráfica.
- Kernel del SO.

3.3.3.2.- Parámetros dinámicos (recopilados cada 30 seg):

- Carga de CPU.
- % RAM ocupada.
- % swap ocupado.
- Tiempos entre caídas del sistema.
- Nº de procesos lanzados.

3.3.4.- Información general del clúster.

Aquí guardamos la información general que servirá también para la asignación de pesos a los parámetros de máquinas y procesos.

- Lista de máquinas.
- Nº de procesos lanzados en todo el cluster.
- Fecha/hora.
- ¿Día festivo?
- Carga del planificador.
- Lista de planificadores.

3.3.5.- Esquemas de tablas de la Base de Datos.

Para un diseño eficiente de la Base de Datos utilizada por el planificador, se implementan las siguientes tablas: (se indican los campos que actúan de clave en negrita).

3.3.5.1.- Tabla de clientes.

Esta tabla contiene a los clientes que se encuentran esperando la ejecución de una orden:

Id. Sesión	IP Cliente	UID	Puerto recepción

Id. Sesión: (Clave). Identificador de la sesión.

IP Cliente: Dirección IP del cliente destinataio de la ejecución de la orden

UID: Identificador del usuario ejecutor de la orden.

Puerto recepcion: Puerto por el que l cliente espera la recepción de la orden.

Esta tabla no resulta muy necesaria, ya que el sistema no necesita los datos de los clientes, salvo por razones informativas, y para mostrar en la GUI.

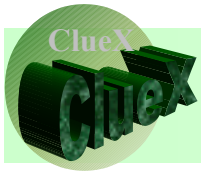
3.3.5.2.- Tabla dinámica de procesos.

Contiene los procesos en ejecución:

GPID	IP Estación

GPID: Identificador global del proceso (identificador que maneja el planificador).

IP Estación: Dirección IP de la estación de cálculo que se encarga de la ejecución del proceso.



3.3.5.3.- Tabla de Estaciones de Cálculo.

En ella figuran las estaciones de cálculo que tienen, al menos, un proceso pendiente de ejecución.

IP Estación	Datos estáticos	Datos dinámicos	Lista GPIDs

IP Estación: Dirección IP de la estación de cálculo encargada de la ejecución del proceso.

Datos estáticos: Conjunto de datos de rendimiento estáticos que son relevantes para los algoritmos de asignación de tareas a estaciones de cálculo que usamos.

Datos dinámicos: Conjunto de factores de rendimiento dinámicos que son relevantes para los algoritmos de asignación de tareas a estaciones de cálculo que usamos.

Lista GPIDs: Lista de identificadores globales de proceso que está ejecutando cada estación de cálculo.

3.3.5.4.- Tabla de Historial de Procesos.

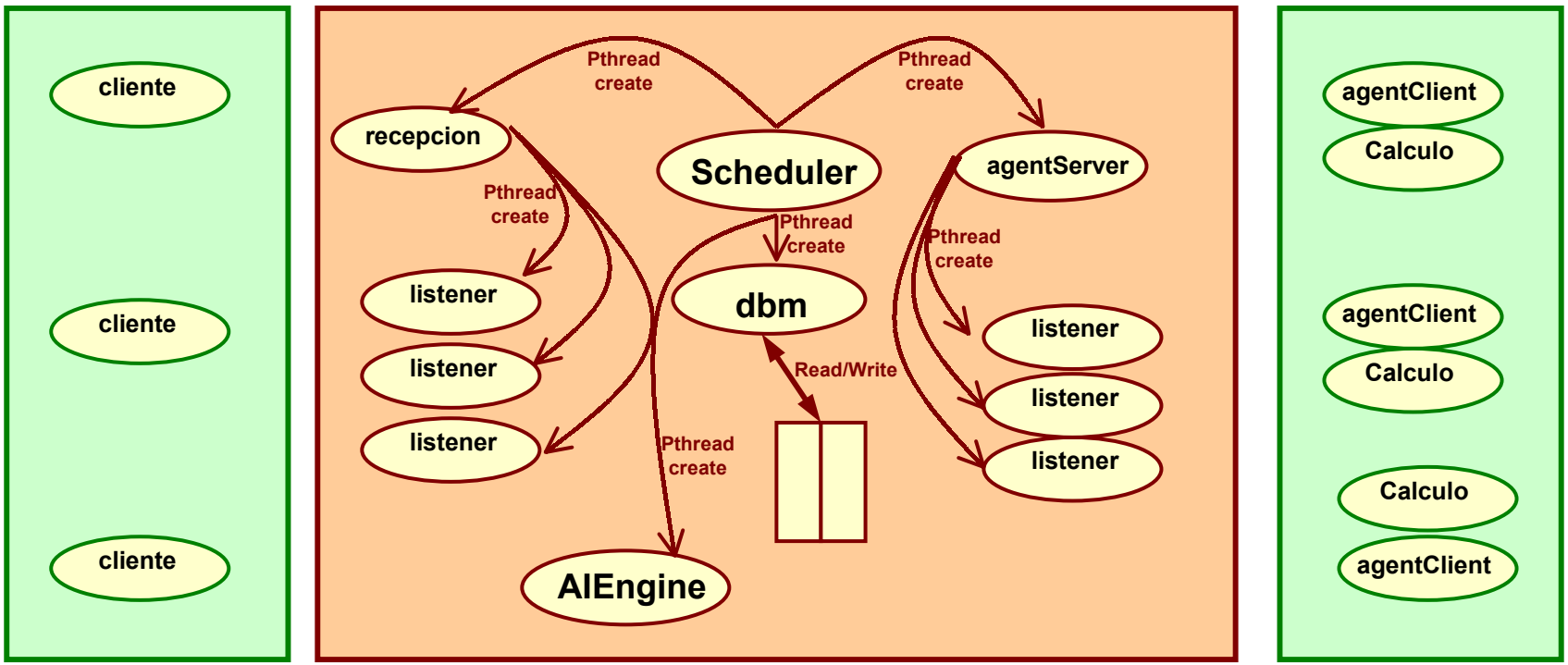
Los datos contenidos en esta table serán muy útiles para los agentes encargados del reparto de tareas en el clúster.

Ruta	Estadísticas

Ruta: Camino completo de ejecutable del proceso.

Estadísticas: Conjunto de datos estadísticos acerca de cada tipo de proceso. Datos relevantes para los algoritmos de asignación de procesos a estaciones de cálculo.

3.3.6.- El Gestor de la Base de Datos.



3.3.6.- El Gestor de la Base de Datos.

3.3.6.1.- Introducción..

- Centraliza el acceso a los datos que maneja el motor de IA de ClueX.
- Implementa el mecanismo para el acceso concurrente a los datos.
- Se basa en el motor de Base de Datos GNU gdbm.
- Las tablas de la BD son tablas hash almacenadas en disco.

3.3.6.- El Gestor de la Base de Datos.

- Tablas del DBM implementadas:
  - Tabla de Estaciones de calculo.
  - Tabla de Procesos activos.
  - Tabla de Historial de procesos (en desarrollo).
  - Tabla de Usuarios del sistema (en desarrollo).

3.3.6.- El Gestor de la Base de Datos.

3.3.6.2.- Tabla de Estaciones de Cálculo.

-Almacena la información estática y dinámica de cada Estación de cálculo (*struct machineinfo*):

- Información de CPU.

  cpuinfo: Mhz, cache, MIPS, arquitectura.
  dinamicload: carga de CPU.

- Información del sistema de memoria.

  meminfo: Tamaño RAM, tamaño de Swap;
  dinamicload: Carga de RAM y carga de Swap).

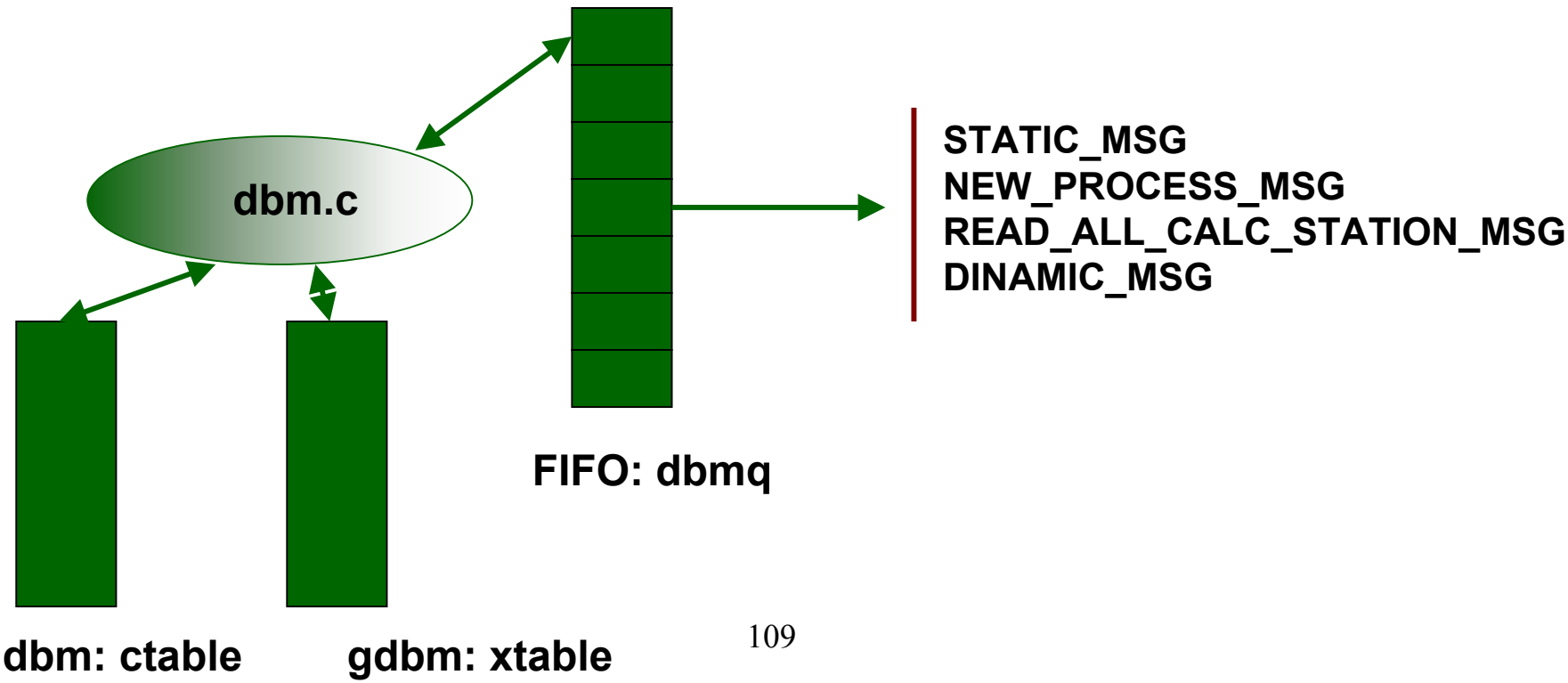
3.3.6.- El Gestor de la Base de Datos.

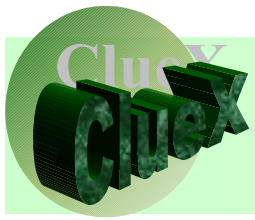
- Información del sistema operativo
  (sysinfo: Versión de kernel, nombre del SO).

- Lista de procesos (GPIDs) en ejecución.

3.3.6.- El Gestor de la Base de Datos.

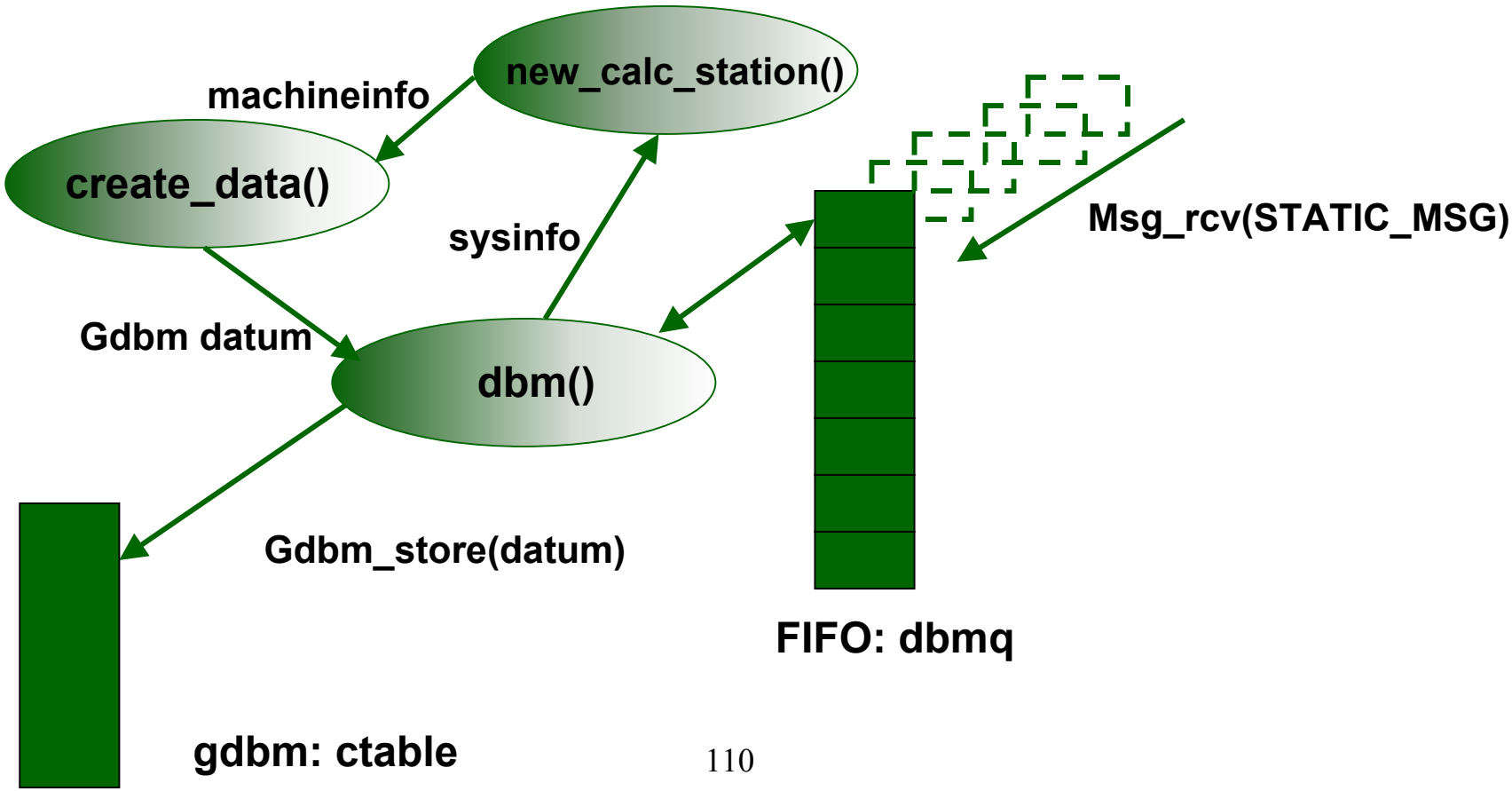
3.3.6.3.- Estructura de la tabla de Estaciones de Cálculo.





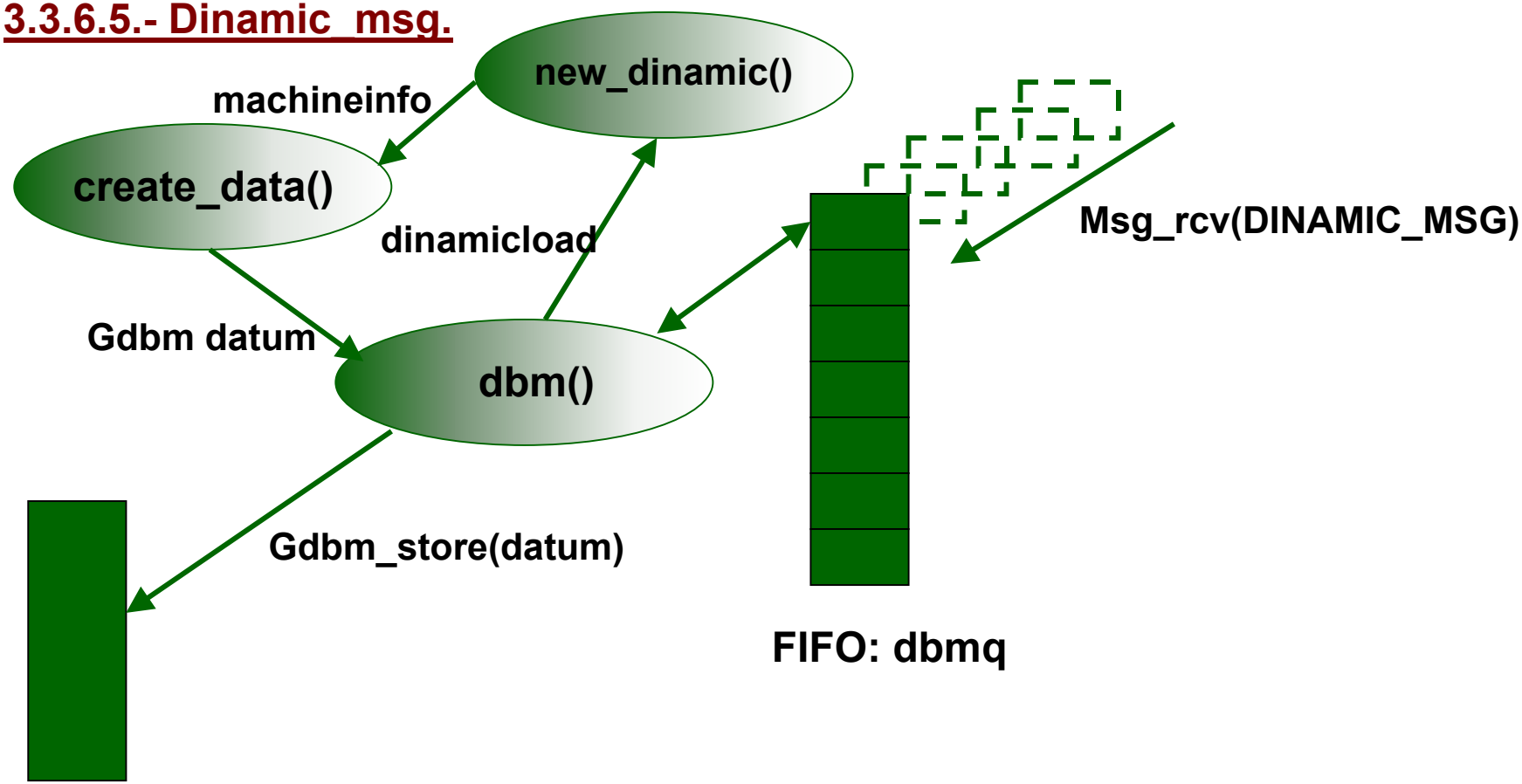
3.3.6.- El Gestor de la Base de Datos.

3.3.6.4.- Static_msg.



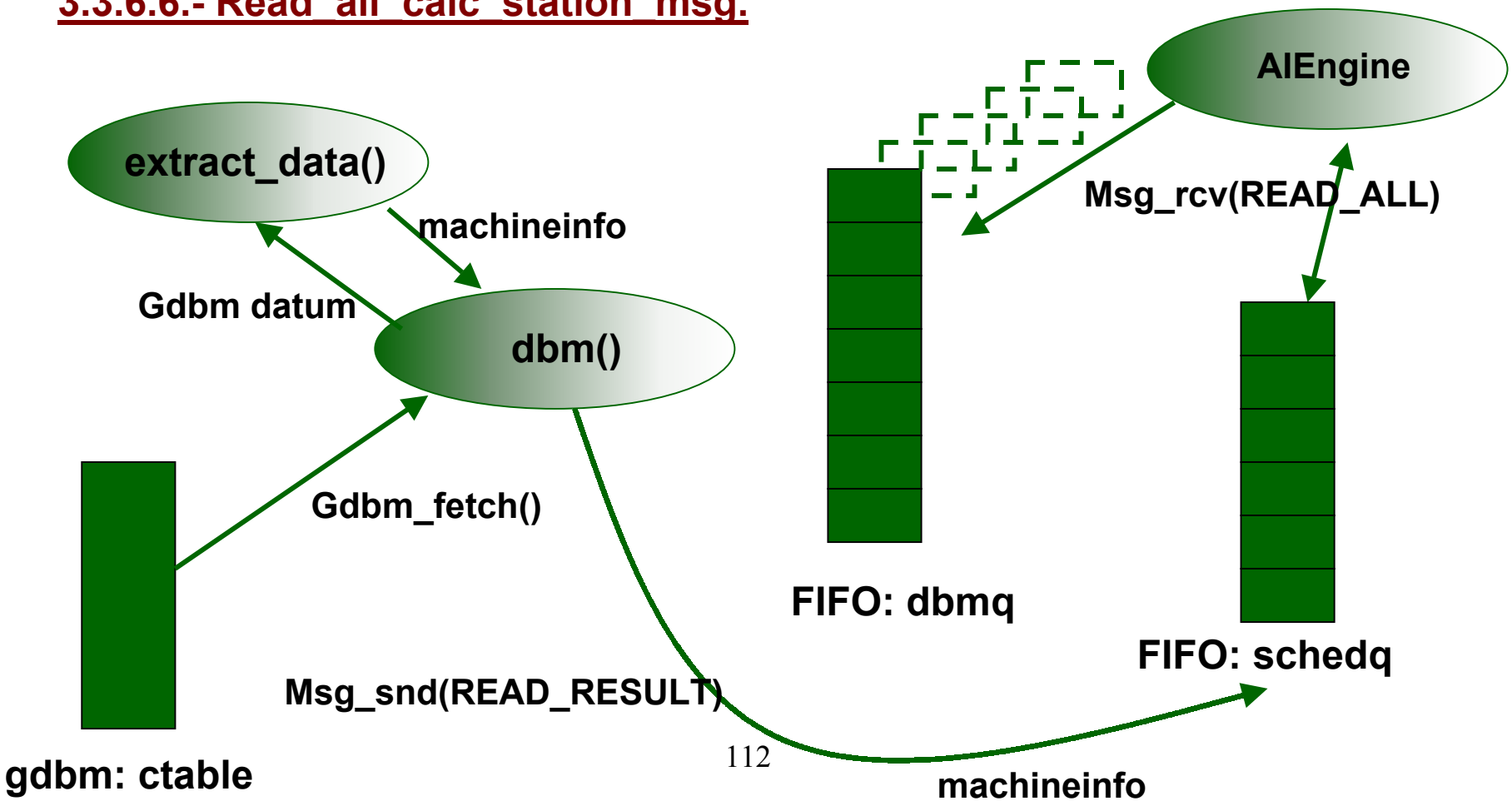
3.3.6.- El Gestor de la Base de Datos.

3.3.6.5.- Dinamic_msg.



3.3.6.- El Gestor de la Base de Datos.

3.3.6.6.- Read_all_calc_station_msg.



3.3.6.- El Gestor de la Base de Datos.

3.3.6.7.- Centralización de funcionalidad.

- Como se puede ver, dbm() es una función de dispatching a la espera de mensajes provenientes de *AIEngine* y de *agentServer* (y proximamente de GUI Applet).

- Una vez que se recibe un mensaje, se llama a la función correspondiente.

3.3.6.- El Gestor de la Base de Datos.

- Para la conversión de datos usamos *create* y *extract data*, donde centralizamos todas las conversiones de datos para todas las tablas.

- De esta forma conseguimos flexibilidad:

  • En funcionalidad:

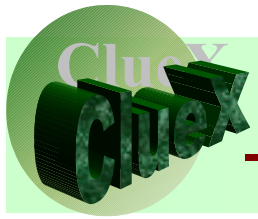
    Para nuevas funciones sólo tenemos que añadir un nuevo tipo de mensaje sin modificar el resto del código.

  • En proceso de datos:

    Para nuevos datos, sólo habrá que modificar *create* y *extract data*.

MÓDULO IV

ALGORITMOS DE PLANIFICACIÓN



4.1.- El motor de IA de ClueX: AIEngine.c

4.1.1.- Introducción.

- Centraliza el mecanismo de asignación de máquina para cada proceso que va a entrar en ejecución.

- Atiende peticiones desde los listener de sched_recepcion.c en una cola de mansajes.

- Aquí es donde tenemos el punto crítico de rendimiento, ya que, dependiendo de lo que tarde en decidir el sistema y lo coherente de sus decisiones, tendremos un balanceo más o menos estable.

4.1.- El motor de IA de ClueX: AIEngine.c

Algoritmos de balanceo:

- Round Robin.
- Case Based Reasonig .
- Redes Neuronales.
- Balanceo de carga.

4.1.- El motor de IA de ClueX: AIEngine.c

4.1.2.- La variable *sched_policy*.

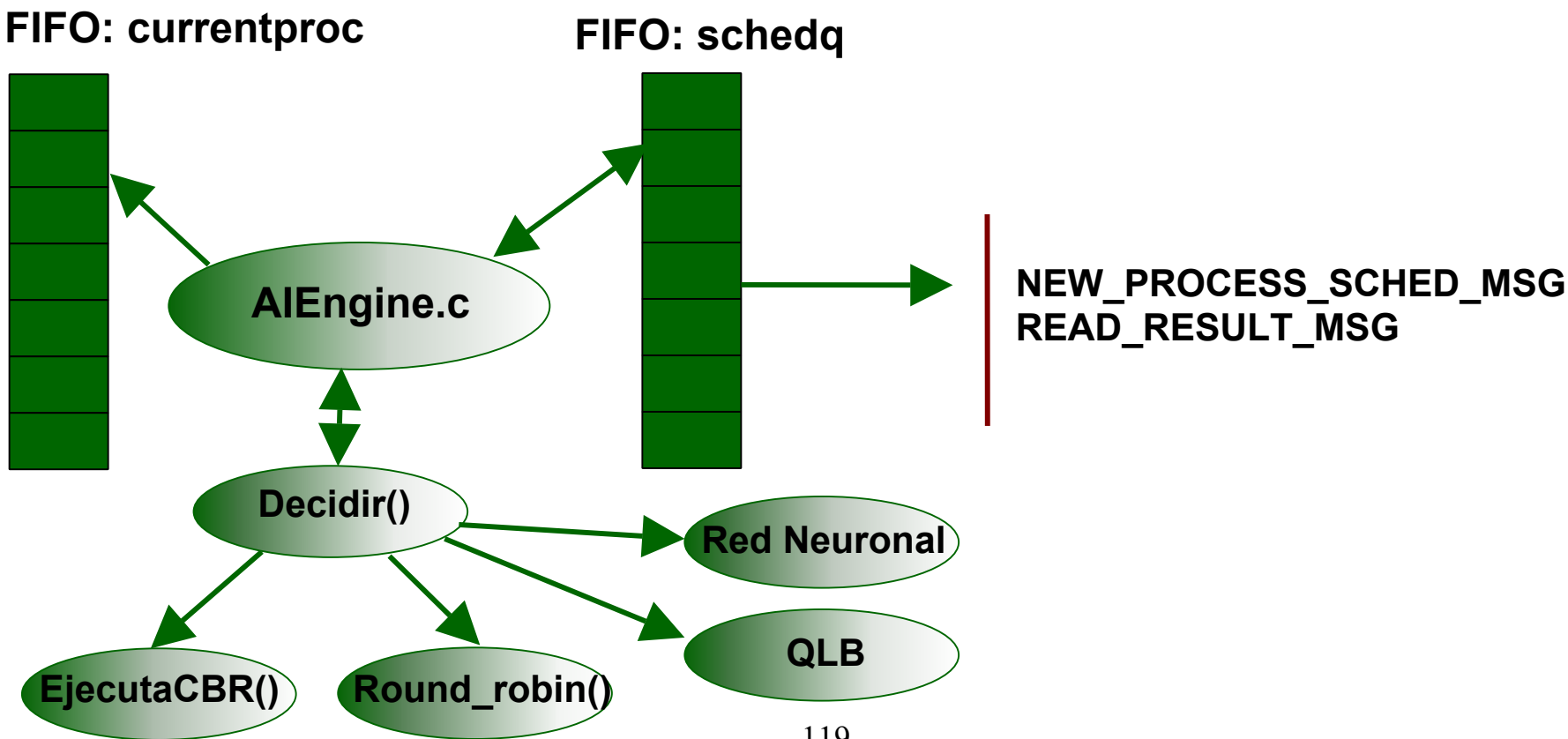
- Indica el algoritmo de planificación a seguir.

- Nos proporciona flexibilidad a la hora de implementar nuevos algoritmos.

- Se usa en *decidir()* de tal forma que para incrustar un nuevo algoritmo, sólo habrá que diferenciar un valor más de *sched_policy* y llamar a una función que arranque el proceso de decisión.

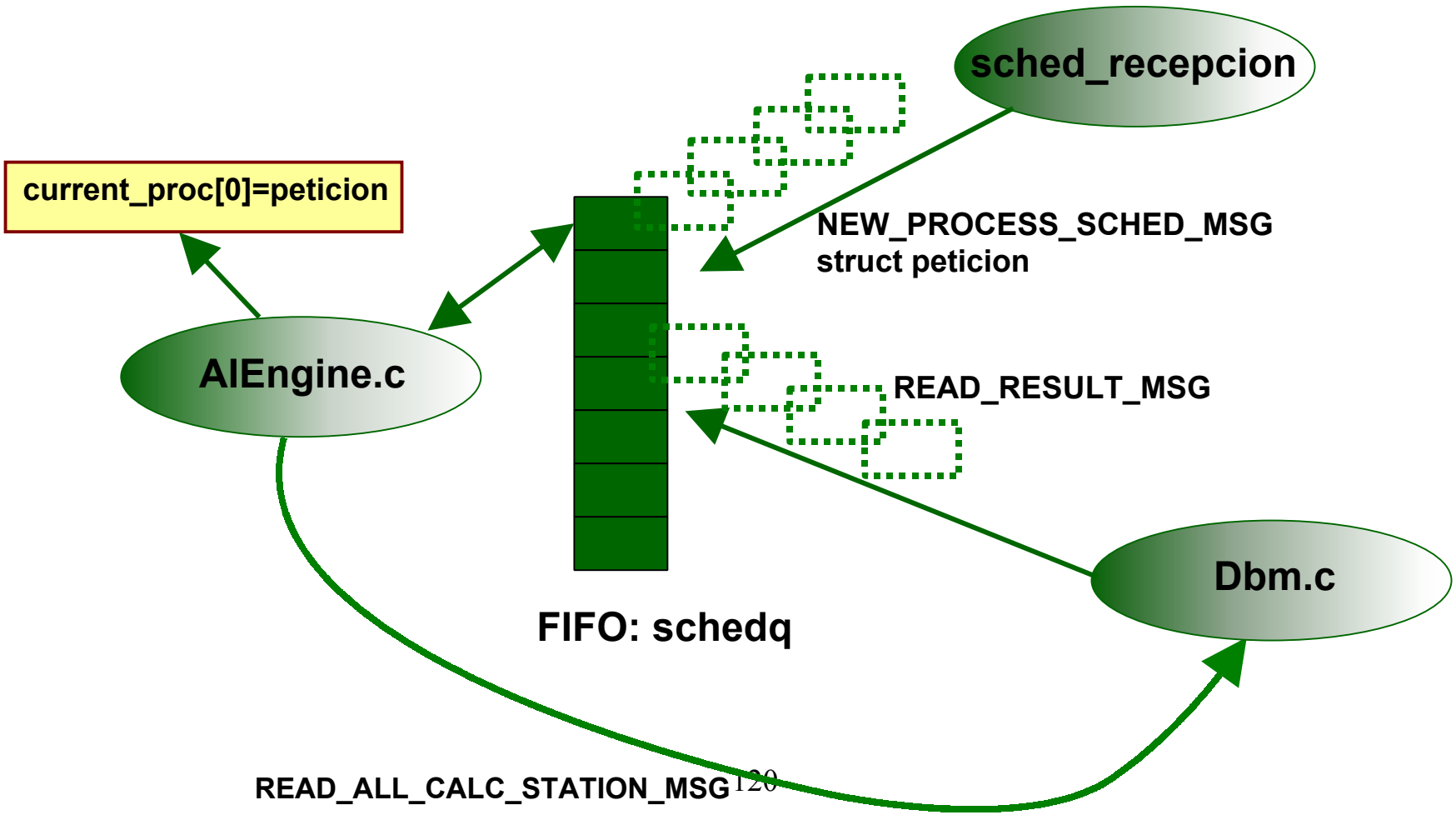
4.1.- El motor de IA de ClueX: AIEngine.c

4.1.3.- Estructura.



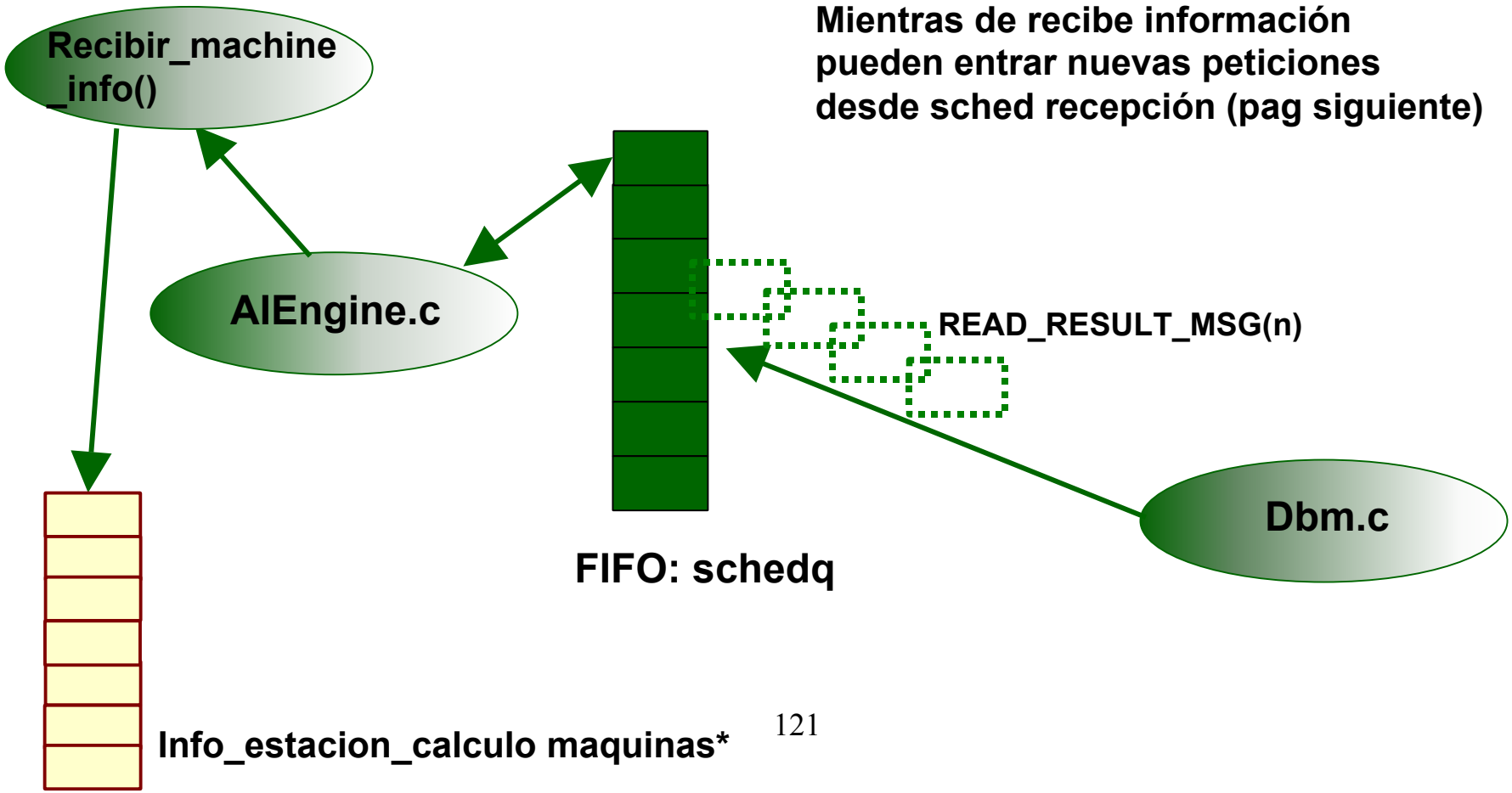
4.1.- El motor de IA de ClueX: AIEngine.c

4.1.4.- *New_process_msg* (1).



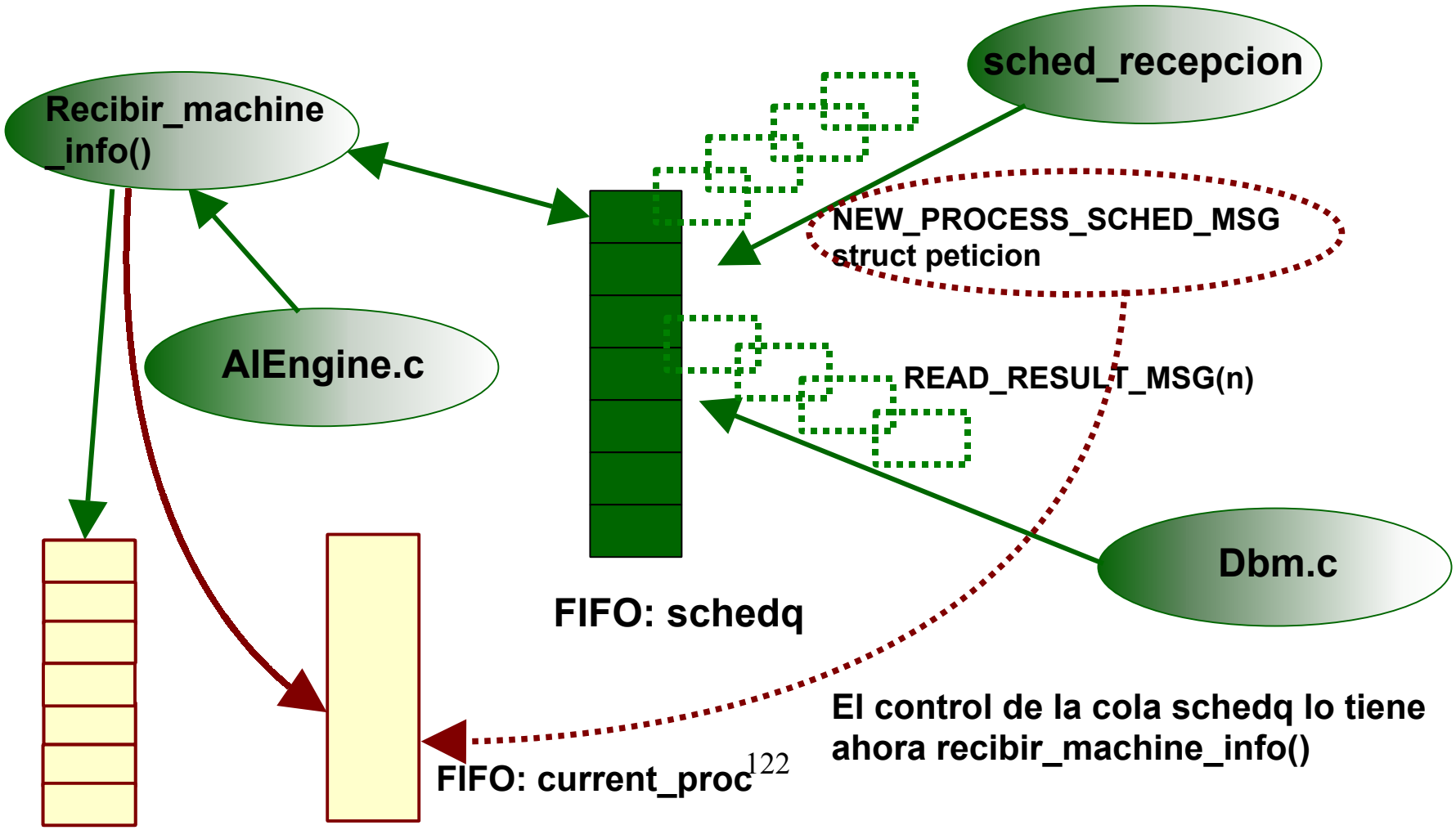
4.1.- El motor de IA de ClueX: AIEngine.c

4.1.4.- *New_process_msg* (1).



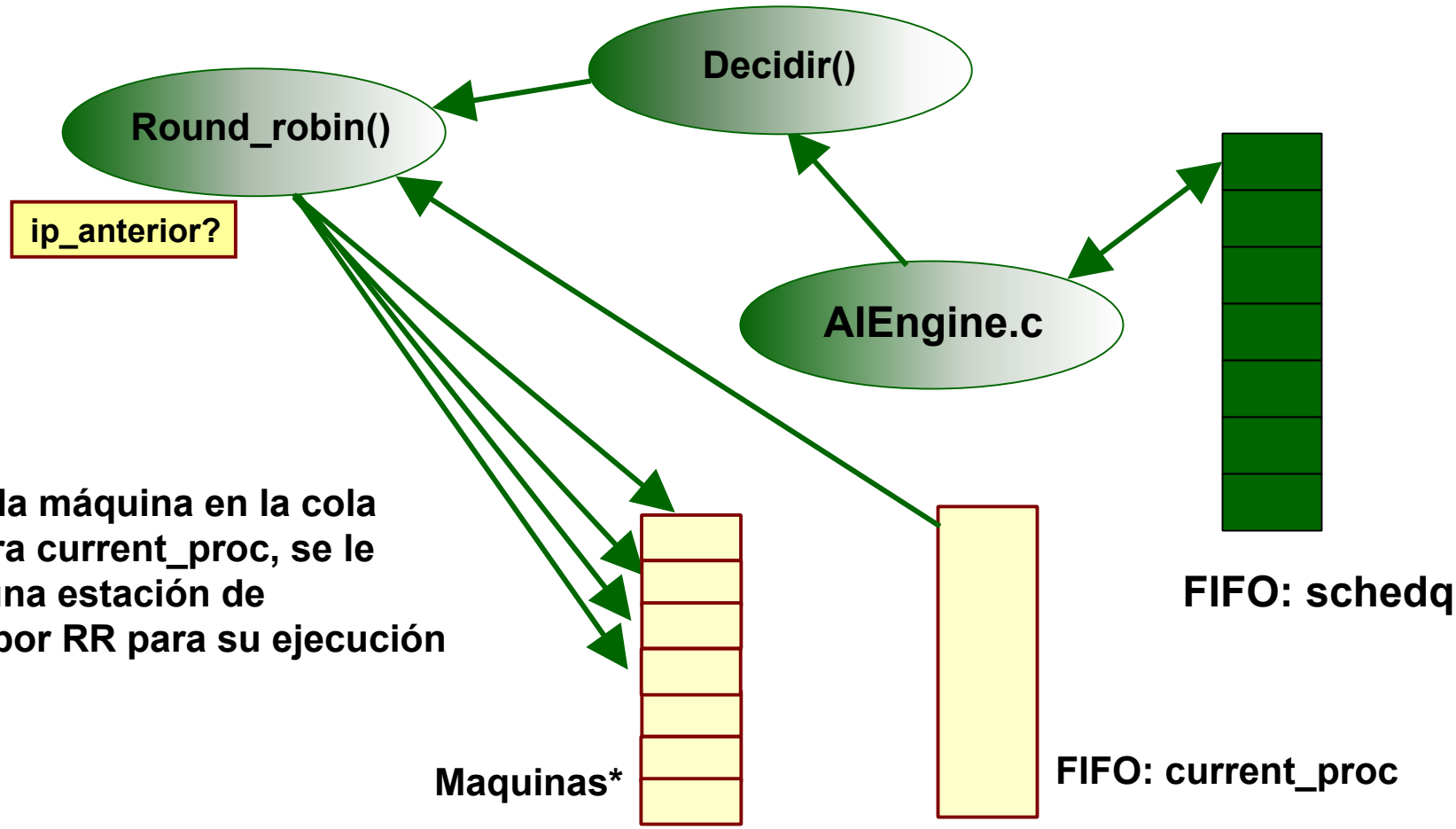
4.1.- El motor de IA de ClueX: AIEngine.c

4.1.5.- *New_process_msg* (n).



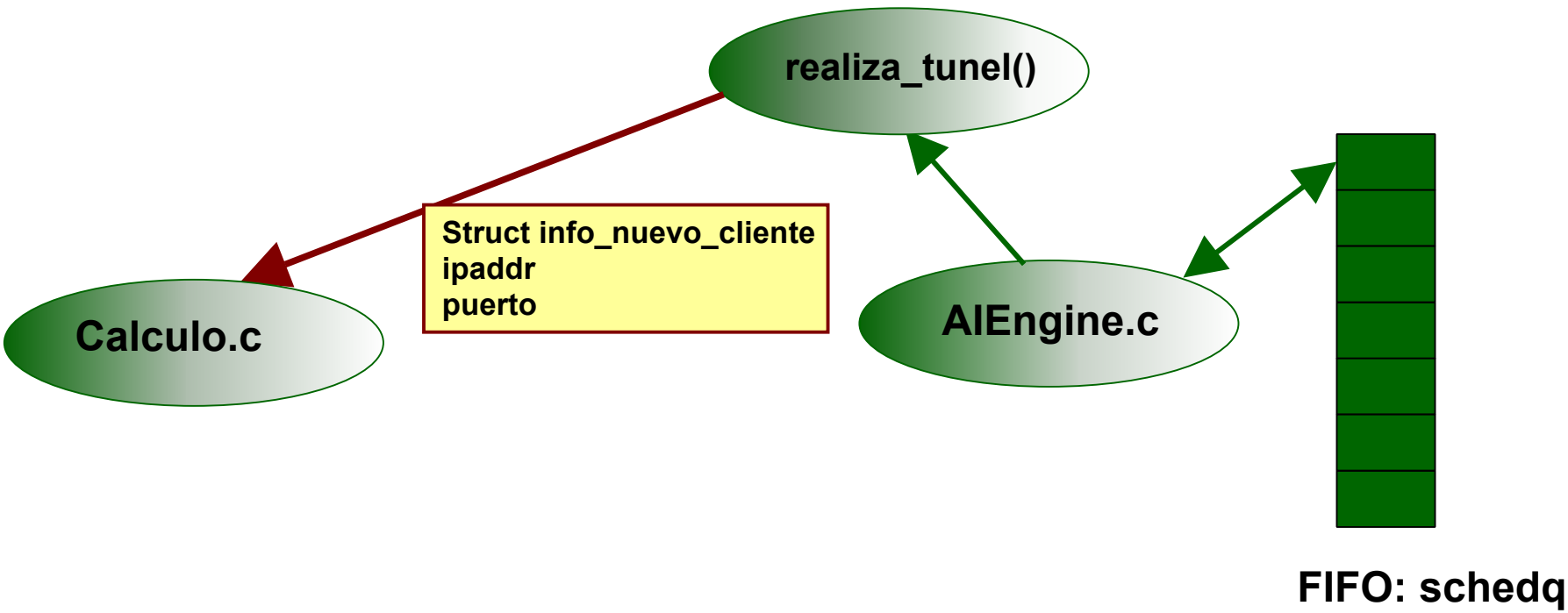
4.1.- El motor de IA de ClueX: AIEngine.c

4.1.6.- Decisión.



4.1.- El motor de IA de ClueX: AIEngine.c

4.1.7.- Comunicación con la estación de cálculo.



4.1.- El motor de IA de ClueX: AIEngine.c

4.1.8.- *New_process_sched_msg.*

- Cuando llega el primer mensaje de este tipo, se pide al dbm toda la información sobre las estaciones de cálculo.
- Como este proceso puede tardar, no queremos dejar desatendida schedq, con lo que seguimos escuchando mensajes *NEW_PROCESS_SCHED_MSG*.
- Si llega un nuevo mensaje cuando estoy recibiendo la información del dbm, lo ponemos en currentproc[i], que es la cola de procesos en espera de ejecución.

4.1.- El motor de IA de ClueX: AIEngine.c

- Cuando se terminan de planificar todos los procesos de la cola *current_proc*, se vacía y se vuelve al bucle normal de dispatching de mensajes.

- (En desarrollo). Para no leer continuamente información del dbm, asignamos un temporizador de tal forma que si han pasado menos de 30 seg desde la última lectura, se usa la información antigua.

4.1.- El motor de IA de ClueX: AIEngine.c

4.1.9.- Otros aspectos.

- Rendimiento con respuesta rápida.
  - Implementado por la cola de procesos en espera.

- Rendimiento por proceso de datos.
  - Implementado por el temporizador.

- Puntos básicos para resto de algoritmos.
  - La base de datos usada para CBR debe ser separada de la de ClueX DBM, para evitar colapsar las colas de mensajes.

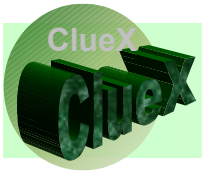
Ciertamente conocido, el Round Robin (Turno Rotatorio), consiste en un **algoritmo ciego** (es decir, no utiliza ningún tipo de heurística), que reparte de forma regular y equitativa los trabajos a las estaciones de cálculo.

Cada tarea es, por tanto, destinada a la estación de cálculo que “le toque”, independientemente de si hubiese sido mejor destinarla a otra máquina desde cualquier punto de vista.

A continuación, podemos ver el **fragmento de código** que implementa el reparto: (para más detalle, consultar el código; concretamente, el módulo “AIEngine”):

```
if (ip_anterior == 0)
{
      ip_anterior = maquinas[0].ip_addr;
      return maquinas[0].ip_addr;
}
else {
      i = 0;
      seguir = 1;
      while((i<tamanio) && (seguir))
      {
            if (maquinas[i].ip_addr == ip_anterior)
                  seguir = 0;
            i++;
      }
      ip_anterior = maquinas[i%tamanio].ip_addr;
      return ip_anterior;
      }
```

Donde “**maquinas**” es el array que contiene la información de las estaciones de cálculo. (Concretamente, se selecciona la IP, para establecer la comunicación).



4.3.- Implementación del sistema CBR.

4.3.1.- Introducción.

Se trata de realizar un **algoritmo basado en casos** para el reparto de tareas entre varios PCs conectados entre sí.

De los PCs tenemos conocida la información estática (velocidad, tamaño de RAM...) y dinámica (carga actual de CPU, memoria libre, transferencias de E/S...). También son conocidos los tiempos de ejecución anteriores de los procesos. Se intenta que el reparto consiga la mayor eficiencia posible en tiempo y recursos.

Dado un proceso a ejecutar, el algoritmo deberá ser capaz de decirnos la máquina a la que se debe asignar.

4.3.2.- Representación de los datos.

Dado que la elección de tipos de datos adecuados para el sistema CBR es muy compleja debido a las necesidades de rendimiento, iremos exponiendo posibilidades que más adelante son probadas y refinadas.

4.3.2.1.- Aproximación 1.

El sistema en que se enmarca el CBR nos obliga a utilizar **tablas hash** para implementar la Base de Datos. Inicialmente, tenemos un **árbol formado por 2 niveles**:

- **Raíz:** tabla hash con una entrada por comando. El nombre del comando serviría de clave, y contiene la tabla de ejecuciones pasadas de dicho comando.

- **Hojas:** tabla de ejecuciones pasadas de un comando determinado. La representación más evidente de las hojas sería:
  - **Nombre** (ip) de los PCs del cluster en la cabecera.
  - En cada fila: el **estado del PC** en el momento de la ejecución (% de CPU, cantidad de memoria, estado E/S, etc.), tiempo de ejecución del comando, y PC elegido para la ejecución.

PC #1	PC #2	PC #3	PC #4	...	PC #N	Tiempo ejecución	PC elegido
CPU 90% MEM 80% I/O 30%	CPU 40% MEM 50% I/O 70%	CPU 40% MEM 50% I/O 50%	CPU 20% MEM 50% I/O 70%		CPU 90% MEM 80% I/O 30%	120 ms	3
CPU 40% MEM 60% I/O 70%	CPU 80% MEM 40% I/O 20%	CPU 50% MEM 70% I/O 10%	CPU 50% MEM 30% I/O 20%		CPU 90% MEM 40% I/O 30%	50 ms	2
CPU 50% MEM 30% I/O 30%	CPU 90% MEM 40% I/O 20%	CPU 20% MEM 80% I/O 10%	CPU 50% MEM 50% I/O 30%		CPU 50% MEM 60% I/O 30%	780 ms	1
CPU 70% MEM 40% I/O 20%	CPU 40% MEM 30% I/O 30%	CPU 50% MEM 40% I/O 30%	CPU 50% MEM 20% I/O 20%		CPU 70% MEM 80% I/O 20%	80 ms	4
...	...	...	...		…	…	…

El algoritmo consistiría en elegir en la raíz la tabla del comando a ejecutar. Si hay suficientes casos en la tabla, compararíamos con el estado actual de las CPUs, eligiendo el caso que mejor encajara.

Si no hubiera casos suficientes, elegiríamos un PC aleatoriamente o mediante un algoritmo sencillo como round-robin.

4.3.2.2.- Aproximación 2.

La estructura de tabla de las hojas, no es válida porque **el número de máquinas en el cluster puede variar**, y ello nos obligaría a tener que **cambiar la cabecera**. Una representación que solucionaría este problema sería:

- **Cabecera:** %CPU, Memoria, I/O, etc divididos en intervalos .
- **Filas:** Número de PCs que cumplen el estado de la cabecera en el momento de la ejecución del comando.

CPU 90% MEM 90% I/O 90%	**CPU 90% MEM 90% I/O 80%**	**CPU 90% MEM 90% I/O 70%**	**CPU 90% MEM 90% I/O 60%**	**……**	**CPU 10% MEM 10% I/O 10%**	**Tiempo ejecución**	**PC elegido**
0	1	2	0		0	120 ms	2
0	2	0	0		1	50 ms	1
1	0	0	2		0	100 ms	3
...	...	...	...		...	...	...

Ventajas: Comparación de casos eficaz (relativamente según se verá más adelante).

Para comparar un caso compararemos las celdas de cada fila y haremos una resta para comprobar si el número de máquinas en el mismo estado coincide.

Esta comparación tiene una complejidad de **O(n)**. Siendo n el número de columnas.

Desventajas: Desperdicio de memoria.

Tomemos como ejemplo que elgimos representar en la cabecera la siguiente información:

Variables incluidas	Intervalos
% ocupación CPU	5 intervalos: 0%,25%,50%,75%,100%
% ocupación Memoria	4 intervalos: 0%, 33%, 66%, 100%
Velocidad CPU	3 intervalos: low-medium-high
Tamaño Memoria	3 intervalos: low-medium-high
Velocidad I/O	3 intervalos: low-medium-high

En total tendríamos: 5x4x3x3x3 = 540 columnas → 540 bytes por fila.

4.3.2.3.- Aproximación 3.

Para evitar el desperdicio de memoria en vez de indicar el número de PCs en cada columna tendríamos un **array** en el que **cada elemento representa un PC y el contenido de ese elemento sería el estado en el que se encuentra.**

Seguimos manteniendo la cabecera de la aproximación anterior, pero en este caso la representación de las filas cambia.

Ejemplo:

Supongamos que tuviéramos en la cabecera de la tabla 7 estados, y hubiera 5 máquinas conectadas al cluster (quitamos la información del tiempo de ejecución y la máquina elegida por simplicidad):

Estado #1	Estado #2	Estado #3	Estado #4	Estado #5	Estado #6	Estado #7

La representación de la aproximación 2 sería:

1	0	2	0	1	1	0

Con la nueva representación la representación tendríamos:

1	3	3	5	6

El contenido de cada celda indica el estado de una máquina. Como tenemos 2 máquinas en el estado 3, aparecen 2 celdas conteniendo un 3 como estado.

Ventaja: Ahorro de memoria.

Suponiendo que la fila tuviera una longitud de m celdas (es decir: m = número de PCs conectados al cluster), y que cada celda ocupe 4 bytes (tamaño utilizado normalmente para los enteros) tendríamos que cada fila costaría:

4xm bytes

Si comparamos la memoria ocupada en la aproximación 2 y en la actual, tenemos que se gasta la misma memoria en el caso de que:

4xm = n
(siendo n la longitud de una fila en la aprox. 2. Es decir, el numero de estados distintos en los que podemos clasificar cada máquina).

Es decir si m < n/4 la nueva opción es más rentable. Aparte sabemos que m<<n, por lo que si nos fijamos en el primer ejemplo donde teníamos una tabla de 540 estados, el número de PCs máximo con el que ahorraríamos memoria sería 135. (135 x 4 = 540).

Este resultado es muy obvio en el sentido de que en el ejemplo de 540 columnas faltaban variables por incluir y los intervalos eran bastante grandes, por lo que a la hora de la implementación, el número de estados aumentará.

En esta situación y suponiendo que el número de PCs no superara el centenar comprobamos que **la nueva aproximación es mucho más rentable que la anterior.**

Desventajas: **Coste de la comparación**.

Para poder comparar cada caso tendríamos que rellenar una estructura como las filas de la aproximación 2 y seguir el método que explicamos anteriormente. Este método es obligado porque con esta representación hasta que no hemos recorrido todo el array no conocemos el número de PCs en cada estado y ese número es necesario a la hora de hacer la comparación.

El nuevo método de comparación implicaría un coste de:

1. **O(n)** para rellenar la estructura con 0's (ya que se utiliza en cada comparación).

2. **O(m)** para rellenar la estructura con los datos de nuestra fila.

3. **O(n)** para comparar campo a campo.

En total obtenemos un orden de **0(2n+m).**

Aunque hemos conseguido un buen aprovechamiento de la memoria el coste de comparación de cada caso es demasiado alto, por lo que buscamos la siguiente aproximación.

4.3.2.4.- Aproximación 4.

Nos centramos en mejorar el coste de comparación. Para ello:

1. Sólo inicializamos el array de tamaño n (llamemosle "**CASO**") a 0 una sóla vez: **O(despreciable).**

2. Rellenamos con los datos de la fila (llamemosle **"ESTADO_MAQUINAS"): O(m).**

3. En vez de recorrer las celdas del array CASO, **recorremos ESTADO_MAQUINAS** y **buscamos en CASO las posiciones que nos indica el contenido de ESTADO_MAQUINAS**. Si accedemos y hay un 0, significa que para ese estado ya hemos hecho la comparación y sino comparamos y ponemos a 0 esa posición: **O(m)**.

El coste resultante es: **O(2m)**. Bastante mejor que el anterior debido a que m<<n.

Ejemplo:

Supongamos que hemos llegado al paso 3 y tenemos la siguiente situacion:

Estado #1	Estado #2	Estado #3	Estado #4	Estado #5	Estado #6	Estado #7

CASO:

1	0	2	0	1	1	0

ESTADO_MAQUINAS:

1	3	3	5	6

4.3.- Implementación del sistema CBR.

Accedemos a **ESTADO_MAQUINAS[1]** → Tenemos que buscar en **CASO[1]**. Como indica que hay 1 máquina comparamos con nuestro caso (que será una fila igual que CASO) y **ponemos F[1] a 0**:

0	0	2	0	1	1	0

Accedemos a **ESTADO_MAQUINAS[2]** → Tenemos que buscar en **CASO[3]**. Como indica que hay 2 máquinas comparamos y **ponemos esa posición a 0:**

0	0	0	0	1	1	0

Accedemos a **ESTADO_MAQUINAS[3]** → Tenemos que buscar en **CASO[3].** Como contiene un 0, eso indica que ya hemos comparado las máquinas que se encontraban en el estado #3, por lo que no hacemos nada.

Etcétera.

4.3.3.- Algoritmo de petición de ejecución de un nuevo proceso.

Cuando llegue un proceso para ser ejecutado tendremos que modificar nuestra estructura de datos según el siguiente algoritmo:

1.- Consultar DBM: pedir lista “sysinfo” (estado actual de todos los procesadores).
2.- Consultar árbol CBR para consultar el proceso.

CASO A: NO EXISTE EL PROCESO

3.- Insertar el nombre del proceso en la tabla de procesos del DBM.
4.- Crear tabla hash “Arbol<nombre_de_proceso>.dbf”.
5.- Seleccionar “al azar” estación de cálculo para realizar la ejecución.
6.- Esperar ejecución y recoger el tiempo de ejecución de la estación de cálculo.
7.- Añadir fila a la tabla “Arbol<nombre_de_proceso>.dbf” con:

- Estado de los procesadores
- tiempo de ejecución
- decisión => índice de la decisión tomada (máquina elegida)
- validado => FALSE
- numPruebas => 1

CASO B: YA EXISTE EL PROCESO

3.- Leer entradas de “Arbol<nombre_de_proceso>.dbf”.
4.- Aplicar función de similitud:

CASO B.1: NO ENCAJA CON NINGUNA ENTRADA:

5.- Seleccionar “al azar” estación de cálculo para realizar la ejecución.
6.- Esperar ejecución y recoger el tiempo de ejecución de la estación de cálculo.
7.- Añadir fila a la tabla “Arbol<nombre_de_proceso>.dbf” con:

- Estado de los procesadores
- tiempo de ejecución
- decisión => índice de la decisión tomada (máquina elegida)
- validado => FALSE
- numPruebas => 1

CASO B.2: ENCAJA CON LÍNEA SIN VALIDAR:

5.- Elegir “al azar” estación de cálculo donde ejecutar la orden.
6.- Ejecutar orden y recoger el tiempo de ejecución de la estación de cálculo:

Si es mejor que el anterior => actualizar la decision en “Arbol<nombre_de_proceso>.dbf”.
Si no => no hacer nada.

7.- numPruebas:= numPruebas + 1
8.- si (numPruebas > CONSTANTE) => Validar

CASO B.3: ENCAJA CON LÍNEA VALIDADA:

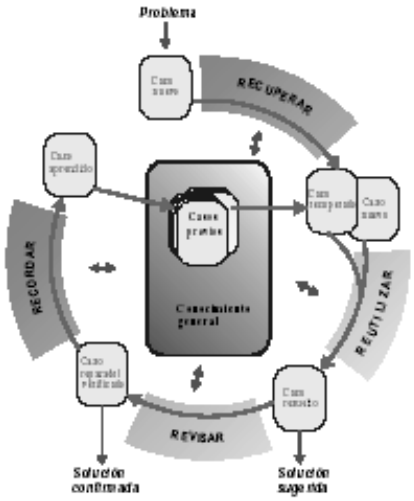
5.- Elegir directamente la máquina validada.

Hay un aspecto del CBR que no se ha señalado con anterioridad pero que aparece implícito en el algoritmo anterior: el **conocimiento del experto** que nos indica qué solución es la correcta.

Como en nuestro caso no es posible (mejor dicho: sería demasiado costoso) determinar cuál sería la mejor máquina, la solución adoptada es ejecutar inicialmente los procesos en máquinas elegidas aleatoriamente. Cuando llevemos cierto número de ejecuciones podremos comprobar que solución (máquina) es la mejor y la validaremos como la solución definitiva para ser usada como caso definitivo en las siguientes ejecuciones del ciclo CBR.

4.3.4.- El ciclo CBR en el algoritmo.

Se puede identificar fácilmente cada una de las etapas del ciclo de un algoritmo basado en casos en el que hemos diseñado para ClueX:

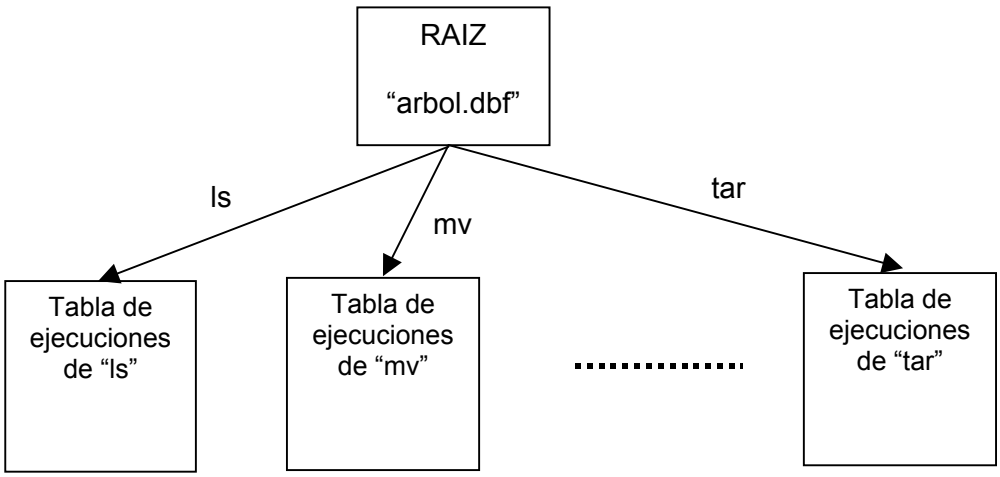


4.3.4.1.- Recuperación.

Realizamos la determinación de los datos relevantes para comparar casos al configurar la cabecera de la tabla de casos, y al definir las prioridades de las características implicadas.

Para encontrar los casos relevantes, la característica principal es el nombre del comando que se quiere ejecutar, recuperando de la Base de Casos todas las anteriores ejecuciones de dicho comando:

Concretamente, en nuestra implementación se ha organizado la Base de Casos como un **árbol de 2 niveles**. En la **raíz** se indexan las tablas donde se guardan las ejecuciones previas de cada comando. **En cada tabla sólo aparecerán las ejecuciones de un comando.**



Una vez obtenida la tabla de las ejecuciones anteriores, se procede a hacer la **comparación** de cada una de ellas con el caso actual, seleccionando el mejor caso según la siguiente función de comparación:

4.3.4.2.-Algoritmo de comparación de casos.

Para terminar de comprender el funcionamiento del algoritmo CBR en nuesto proyecto, sólo falta explicar cuándo consideramos que un caso anterior es lo suficientemente parecido al caso de análisis actual.

A cada una de las características de las máquinas por las que son clasificadas en un "caso" se le ha asignado una **prioridad**, que nos indica la relevancia que tiene a la hora de considerar dos ejecuciones "parecidas". Además, el rango de valores de cada característica lo dividimos en un número determinado de **intervalos** en los que serán clasificadas todas las máquinas, con ayuda de una **función de normalización**.

La ordenación de los estados en la fila que representa un "caso" no se ha realizado de forma trivial: se ha pretendido mantener en posiciones cercanas a los estados parecidos. De esta forma, si no se halla ninguna máquina en el estado que nos interesa, es posible elegir una de las máquinas que se encuentren próximas en el array.

Concretamente, se han mantenido en posiciones consecutivas los estados que difieren entre sí en las características de menor prioridad, manteniendo los mismos valores en las características de mayor importancia.

Veamos un ejemplo en el que manejamos 3 características: A, B y C, siendo A la más prioritaria, y C, la menos prioritaria.

Consideramos que: A tiene 3 intervalos: A_1, A_2 y A_3.
B tiene 4 intervalos: B_1, B_2, B_3 y B_4.
C tiene 3 intervalos: C_1, C_2 y C_3.

Por tanto, representaremos los casos en la siguiente tabla:

$A_1B_1C_1$	$A_1B_1C_2$	$A_1B_1C_3$	$A_1B_2C_1$	$A_1B_2C_2$	$A_1B_2C_3$	$A_1B_3C_1$	...	$A_3B_4C_1$	$A_1B_4C_2$	$A_3B_4C_3$
							...			
							...			
							...			

Buscamos el caso más similar al actual, que será el de **mejor "factor de comparacion"** (cuanto menor sea este valor, mas se parecera un caso al caso actual).

Para ello, en cada caso extraído hacemos lo siguiente: vamos comparando cada una de las máquinas que forman el caso actual , y encontrando la máquina del caso extraido que más se le asemeje. **Cuanto más distintas sean las máquinas (i.e. más distanciadas estén en el array), más aumentará el factor de comparación.**

Una importante observación a la hora de hallar la máquina más parecida a una dada es que **existe la posibilidad de que entre dos posiciones consecutivas del array, la variación del valor de las características sea grande**. En el ejemplo anterior, ocurriría en las posiciones:

$A_1B_1C_1$	$A_1B_1C_2$	$A_1B_1C_3$	$A_1B_2C_1$	$A_1B_2C_2$	$A_1B_2C_3$	$A_1B_3C_1$	...	$A_3B_4C_1$	$A_1B_4C_2$	$A_3B_4C_3$
							...			
							...			
							...			

(1) (2) (3)

Entre las posiciones anterior y posterior a (1), no hay una simple variación de una de las características, sino que hay dos variaciones:

- Variación de la característica B (de B_1 a B_2).
- Variación **grande** de la característica C (de C_3 a C_1).

Por esta razón, **entre este tipo de "posiciones-frontera", no buscamos máquinas similares en nuestro algoritmo.**

Para distinguir estas posiciones frontera, nos ha sido útil la construcción de un **array que hemos llamado "bases"**, y que **nos indica el número de intervalos en los que dividimos el rango de valores de cada característica**, **ordenados por orden de prioridad**. En el ejemplo, dicho array tendría el siguiente contenido:

BASES

3	4	3

Así, BASES[0] nos dará el número de intervalos en que hemos dividido la característica más prioritaria. (En nuestro ejemplo, A).

Las "posiciones-frontera" vendrán dadas por las expresiones:

BASES[3] x i = Posiciones en los que la característica C sufre un gran cambio;
$i \in [1..(BASES[1] \times BASES[2])]$

BASES[3] x BASES [2] x i = Posiciones en que la característica B sufre un gran cambio, $i \in [1..(BASES[1])]$

BASES[3] x BASES[2] x BASES[1] x i = Posiciones en los que la característica A sufre un gran cambio (no llega a ocurrir dentro del tamaño del array BASES)

4.3.4.3.- Reutilización.

Una vez encontrado el caso extraído más parecido al actual, buscamos en él el estado en que se encontraba la máquina que ejecutó la orden, y seleccionamos una del caso actual que se encuentre en ese estado. En el caso en que no hubiese en el caso actual ninguna estación de cálculo en ese estado, seleccionaríamos aquella que se encontrara en el estado más parecido.

4.3.4.4.- Revisión.

Podemos llegar a la situación en que no se pueda hallar ningún caso compatible con el actual. En tal caso, se elegiría aleatoriamente la máquina de ejecución, tarea de la que se encarga el módulo "AIEngine" (el encargado de invocar a CBR).

4.3.4.5.- Recuerdo.

Una vez recibida la ejecución del proceso, se invocará de nuevo al módulo CBR para almacenar en la base de Casos dicha ejecución. Hay que señalar que el ciclo CBR no se realiza de forma secuencial, debido a que la ejecución de un proceso tarda un tiempo indeterminado.

La forma de almacenar un nuevo caso en la base de Casos esta explicada en el apartado de “Petición de ejecución de un nuevo proceso”, donde se indican los cambios a realizar, según el estado de los casos.

4.4.1.- Introducción.

Las redes neuronales son una herramienta de análisis estadístico que permite construir un modelo de comportamiento a partir de un número elevado de ejemplos. Estos ejemplos, a su vez, suelen tener un gran número de atributos.

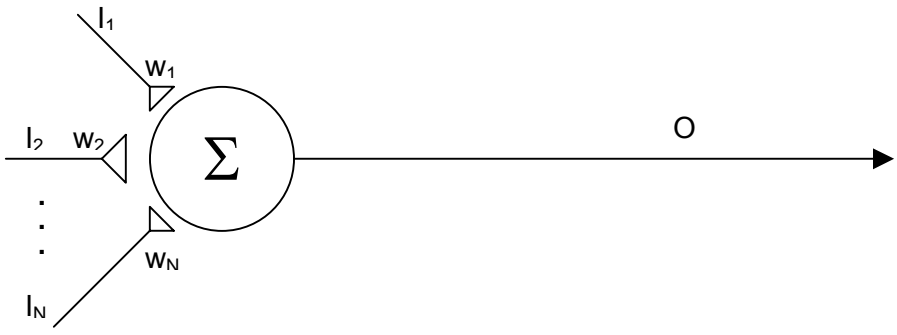
Como es de imaginar, se inspiran en una versión simplificada del modelo del cerebro humano: una red neuronal esta formada por un número de **nodos interconectados**. A las conexiones entre ellos, se les asigna un **peso**, en función de su relevancia para lograr resultados óptimos. Precisamente, existe una fase de aprendizaje que se centra en determinar el mejor valor de esos parámetros.

Las redes neuronales se suelen usar en problemas en los que pueda haber ruido en los datos de entrada, en los que es permisible un tiempo de entrenamiento largo y en los que no se conozca la función objetivo del problema.

En este proyecto, hemos diseñado un algoritmo basado en redes neuronales para que el planificador de ClueX escoja la estación de cálculo más adecuada para una cierta tarea. Como veremos más adelante, hemos aplicado el llamado **"algoritmo de retropropagación"**, para realizar la fase de aprendizaje.

4.4.1.1.- Esquema de una neurona.

Para la neurona j:



Como puede observarse en la figura, cada neurona posee una serie de entradas I_1 ...I_N . A cada una de esas entradas se les asigna sus correspondientes pesos w_1 ...w_N.

En función de los pesos asignados, y de las entradas recibidas en la neurona, y utilizando la llamada **función de activación**, se averigua si la neurona es excitada y genera una salida. Si es así, aplicando la **función de transferencia**, se obtiene el valor de salida.

Normalmente, la función de activación no es más que una suma ponderada de los valores de entrada de las neuronas. Es decir:

$$\sum_{i=1}^{N} w_{ij} x_j$$

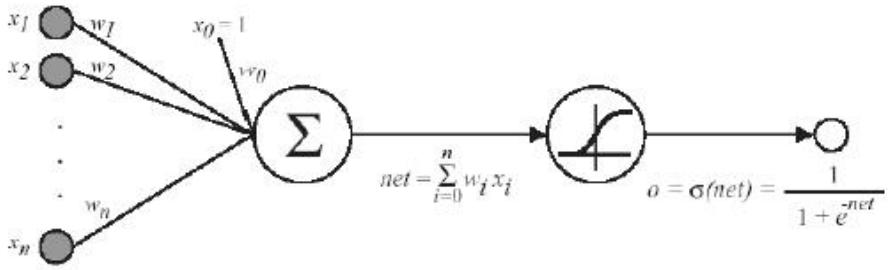
Es posible que el modelo simulado requiera que las neuronas tengan un valor umbral θ. Si es así, la función de activación sería del estilo:

$$\sum_{i=1}^{N} w_{ij} x_j - \theta_i$$

En conclusión, si tenemos una regla de propgación f, el modelo de la neurona j sería:

$$y_i(t) = f_i\left(\sum_{i=1}^{N} w_{ij} x_j - \theta_i\right)$$

En cuanto a la salida que proporciona la neurona, se ha tomado el criterio de que la función de transferencia tenga forma de **sigmoide:**



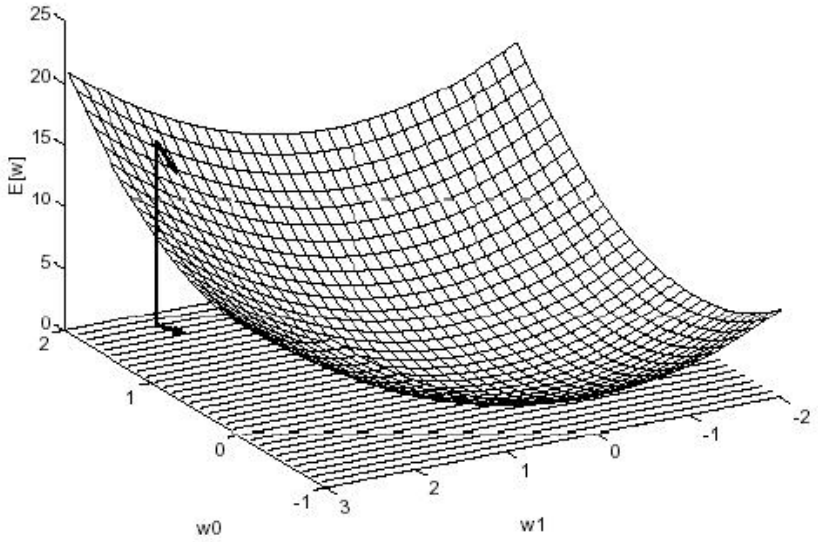
4.4.1.2.- La fase de aprendizaje: descenso por el gradiente.

Se trata normalmente de un proceso iterativo, actualizándose el valor de los pesos una y otra vez, hasta alcanzar el rendimiento deseado.

En nuestro caso, se trata de lograr la optimización de una función coste: proponemos una función error o coste que mide el rendimiento actual de la red, función que dependerá de los pesos sinápticos. Dicha optimización la lograremos con el método de **descenso por el gradiente.**

Para ello, se comienza definiendo una función coste E(.), que proporcione el error actual E que comete la red neuronal, que será una función de los pesos W. Por tanto, E=E(W).

Además, $E : R^n \longrightarrow R$. De esta manera, podemos imaginarnos la representación gráfica de esta función como una hipersuperficie con montañas y valles en la que la posición ocupada por un valle se corresponde con una configuración de pesos localmente óptima, al tratarse de un mínimo local de la función error. El objetivo del aprendizaje será encontrar la configuración de pesos que corresponde a la función global de la función error.



Para encontrar la configuración de pesos óptima se opera del siguiente modo: se parte en t=0 de una configuración de pesos inicial W(0), y se calcula el sentido de la máxima variación de la función E(W) en W(0), que vendrá dado por su **gradiente en W(0)**. El sentido apuntará hasta hacia una colina del paisaje de la hipersuperficie de E(.). A continuación, se modifican los parámetros W, **siguiendo el sentido contrario al indicado por el gradiente de la función error.** De este modo, se lleva a cabo un

descenso por la hipersuperficie del error, aproximándose en una cierta cantidad al valle, un mínimo. El proceso se itera hasta alcanzarlo. Matemáticamente, se expresa del siguiente modo:

$$W(t+1) = W(t) - \varepsilon \nabla E(W)$$

y

$$\nabla E(w) = \left(\frac{\partial E}{\partial w_0} + \frac{\partial E}{\partial w_1} + \ldots + \frac{\partial E}{\partial w_n}\right)$$

donde ε (que puede ser diferente para cada peso) indica el tamaño del paso tomado en cada iteración, que, idealmente, debe ser infinitesimal. Como una elección de este tipo llevaría a un proceso de entrenamiento extremadamente lento, se toma de un tamaño lo suficientemente grande como para que cumpla el compromiso de rápida actualización sin llevar a oscilaciones. (Una actualización excesivamente grande de los pesos nos llevaría lejos de nuestro objetivo: el mínimo).

4.4.1.3.- La retropropagación.

La arquitectura de la red diseñada es multicapa: esta formada por una capa oculta, y otra de salida. Para entrenar las neuronas de la capa oculta usamos el algoritmo de retropropagación de errores. (El hecho de usar sigmoides en la función de transferencia de las neuronas, nos lo permite).

La retropropagación permite aprender los pesos para una red multicapa con un número de unidades e interconexiones dado.

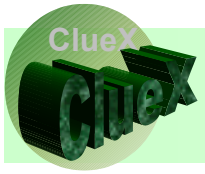
Consideramos una red con múltiples unidades de salida, de forma que el error es la suma de los errores sobre todas las salidas:

$$E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in salidas} (t_{kd} - o_{kd})^2$$

donde: t_{kd} es la salida dada por el ejemplo de entrenamiento d para la unidad k

o_{kd} es la salida generada por la red.

El espacio de hipótesis viene dado por todos los valores posibles para los pesos de todas las unidades de la red.



4.4.- Implementación de la red neuronal.

Se utiliza descenso de gradiente para encontrar una hipótesis que minimice el error aunque sólo se garantiza que se encontrará un mínimo local (suele funcionar bien en la práctica).

El **algoritmo** en sí es el siguiente:

Retropropagacion (ejemplos_entrenamiento, n, num_ent, num_sal, num_ocultas)
Cada ejemplo de entrenamiento es de la forma <x,t>, donde x es el vector de entradas y t el vector de salidas; n es la tasa de aprendizaje (p.e. 0,1), num_ent es el número de entradas de la red, num_ocultas es el número de unidades de la capa oculta y num_sal es el número de unidades de salida; x_{ji} es la entrada que la unidad j recibe de la unidad i, y w_{ji} es el peso de esa entrada.

1.- Se crea una red con num_ent entradas, num_ocultas unidades ocultas, y num_sal unidades de salida.

2.- Se inicializan todos los pesos de la red con números aleatorios pequeños (por ejemplo, entre –0.05 y 0.05)

3.- Mientras que no se cumpla la condición de terminación
...
Para cada ejemplo de entrenamiento <x,t> de ejemplos_entrenamiento:

1.- Se introduce el ejemplo x en la red y se computan las salidas

//Retropropagación de los errores
2.- Para cada unidad de salida k, se calcula su término de error δ_k.

$$\delta_k = o_k(1-o_k)(t_k - o_k)$$

3.- Para cada unidad oculta h, se calcula su término de error δ_h.

$$\delta_h = o_h(1-o_h)\sum_{k \in Salidas} w_{kh}\delta_k$$

4.- Se actualiza cada peso de la red w_{ji}.

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

donde:

$$\Delta w_{ji} = \varepsilon \delta_j x_{ji}$$

El entrenamiento de la red puede requerir miles de iteraciones, aunque la red entrenada es muy eficiente.

La versión mostrada utiliza el descenso de gradiente incremental; para aplicar el descenso puro habría que sumar los valores de $\delta_j x_{ji}$ sobre todos los ejemplos de entrenamiento antes de actualizar los pesos.

Una **variante** habitual es hacer que la actualización de los pesos en la iteración *n* dependa de la actualización en la *n-1*:

$$\Delta w_{ji}(n) = \varepsilon \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

El término adicional representa la **cantidad de movimiento.**

Podemos hallar un símil físico en una pelota que cayera por la superficie de error. La cantidad de movimiento hace que la pelota tienda a mantener la misma dirección:

- Se intenta evitar que la pelota se detenga en un mínimo local.
- Se intenta evitar que la pelota se detenga en un llano.

4.4.2.- La red neuronal de ClueX

La red neuronal implementada en ClueX **sigue exactamente el mismo esquema** que el mostrado en el apartado anterior, incluyendo cantidad de movimiento.

La dificultad más importante al abordar la implementación consistió en conseguir los ejemplos de entrenamiento para poder entrenar la red. Resultaba absolutamente inviable que un "experto humano" indicara al sistema cual era la mejor solución para la planificación de un conjunto de estaciones de cálculo, por lo que el módulo que implementaba la Inteligencia Artificial debía de arreglárselas para conseguir dicha información.

La solución adoptada consistía **en implementar un módulo paralelo al principal que se encargara de enviar un mismo comando a todas las estaciones de cálculo y que éstas le contestaran con los tiempos de ejecución**. De ésta manera tendríamos un vector de la misma longitud que el número de estaciones y cuyos valores serían el tiempo que tardó el comando en ejecutarse en cada una de las

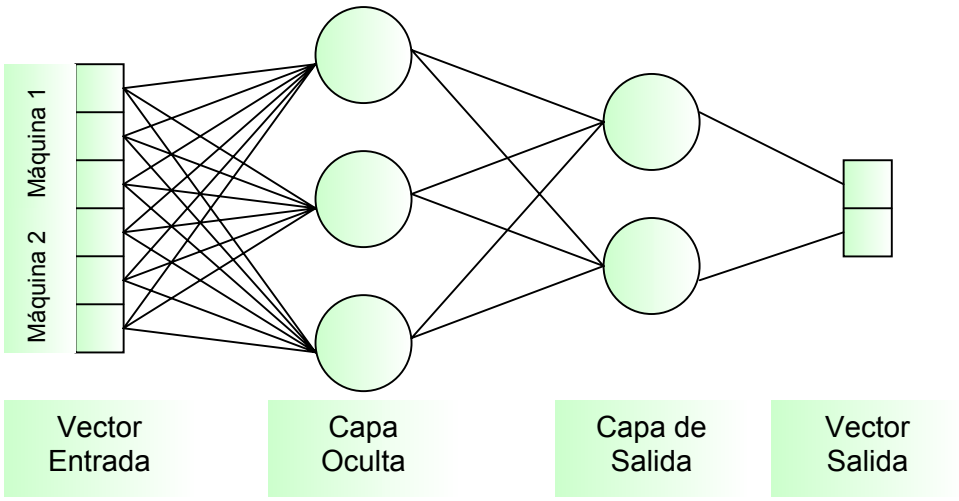
máquinas. Una vez obtenido éste vector podríamos normalizarlo y entrenar la red neuronal con él.

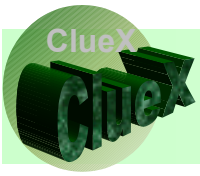
Otro aspecto a decidir residía en fijar la información de las máquinas con las que alimentar la red. Este problema tuvo una solución más intuitiva debido a que las características estáticas de las máquinas no variaban con el tiempo y quedaban implícitas en el vector solución de los tiempos de ejecución, por lo **que sólo había que tener en cuenta las características dinámicas de cada una de las estaciones de cálculo.**

De esta forma, la topología de la red quedaba definida con un vector de entrada formado por los datos dinámicos de las máquinas, un vector de salida con longitud igual al número de máquinas y un vector solución definido por los tiempos de entrenamiento. **La capa oculta constaría de un número predeterminado de neuronas (por defecto 3) y la capa de salida con tantas neuronas como máquinas.**

Los datos dinámicos que forman la entrada de la red constan de la media de utilización de CPU, memoria y SWAP, por lo que el vector de entrada tiene una longitud de tres veces el número de máquinas. (Si se añadieran otras características dinámicas, el diseño permite adaptar la red mediante la modificación de una constante).

Como ejemplo, mostramos la topología de la red para dos máquinas:





4.4.2.1.- Diseño e implementación de la Red Neuronal

Respecto al diseño del código que realiza todas la tareas de la red neuronal, partimos del módulo del AIEngine encargado de dirigir el proceso de la toma de decisiones. Inicialmente se pensó en incluir dentro de dicho proceso el código que manejaba la red neuronal, pero cuando se finalizó la implementación de ésta, resultó ser demasiado compleja para mezclar ambas funcionalidades en un solo módulo. Por eso **se decidió separar la gestión de la red en un proceso aparte que se comunicara con el AIEngine a través de mensajes y zonas de memoria compartida**. Dicho proceso arrancaría cuando la política de planificación cambiara a red neuronal y sería finalizado cuando esta política acabara.

La red neuronal **se implementó utilizando C++**, ya que su diseño prácticamente lo imponía, y se definieron los siguientes métodos de acceso (obviando parámetros):

- *InicializarRedNeuronal().*
- *FinalizarRedNeuronal().*
- *EjecutarRedNeuronal().*
- *EntrenarRedNeuronal().*
- *AjustarRedNeuronal().*
- *GuardarPesos().*
- *RecuperarPesos().*

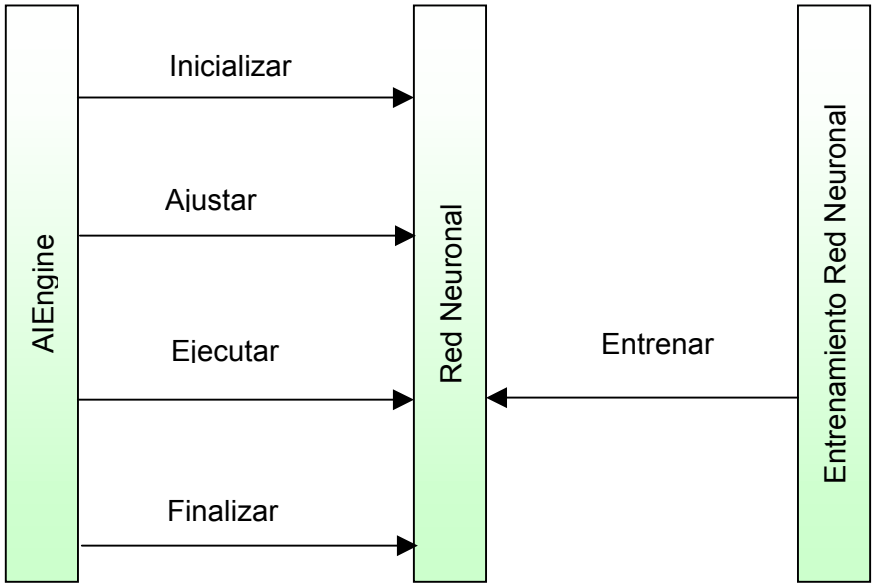
Un método que llama la atención es el encargado de **ajustar la red neuronal**. Su razón de ser se debe a que **la topología de la red cambia** según se añadan o retiren máquinas del cluster, por lo que cuando esto ocurra **habrá que rehacer la red** para ajustarla al nuevo número de estaciones de cálculo. Esta tarea es delicada debido a que el reajuste se podría realizar concurrentemente con un entrenamiento o ejecución por lo que las primitivas mostradas se protegen con un semáforo (IPC de sistemas Unix) para evitar accesos concurrentes no deseados.

Cabe pensar que no sería necesario rehacer toda la red cuando se añada una máquina y que bastaría con añadir la neurona correspondiente al final de la capa de salida y enlazar el nuevo tamaño del vector de entrada con la capa oculta. Sin embargo esta solución, aunque implementada inicialmente se descartó. Como argumentación exponemos un ejemplo: Si tuviéramos una red entrenada donde sus pesos se hubieran acercado a un valor extremo del rango ([-1.1]), por ejemplo al 1, y en ese momento se añadiera una estación que los cambiara al otro extremo, la etapa de entrenamiento se alargaría mucho más que si todos los pesos se reiniciaran cerca del 0 (la inicialización de la red los reparte entre –0.05 y 0.05). Por ese motivo se descartó dicho tipo de ajuste y se implementó el consistente en reiniciar todos los pesos.

4.4.- Implementación de la red neuronal.

El reajuste de la red lo llevará a cabo el módulo AIEngine cuando se le indique que hubo un cambio en el número de estaciones. Dicho módulo sólo utilizará las primitivas de inicialización, finalización, ejecución y ajuste. El resto de métodos los utilizará el módulo encargado del entrenamiento.

Concurrentemente al proceso encargado de gestionar la red neuronal se ejecutará el proceso encargado del entrenamiento:



La implementación de la red permite ajustar las variables explicadas en el resumen teórico de los apartados anteriores como la tasa y el momento. Por defecto toman un valor de 0.3, pero pueden ser modificados dentro del archivo configuración *nnetconfig.xml*

Respecto a la toma de decisión, si dejamos a un lado el entrenamiento**, el módulo AIEngine sólo se tiene que encargar de alimentar la red con los datos dinámicos del momento de la decisión (proporcionados por el dbm) y recoger la salida del procedimiento** *EjecutarRedNeuronal().* Este método devuelve la máquina a la que enviar el comando, y si surgiera algún error se indicaría de forma que se ejecute round robin para la decisión. Los errores a los que nos referimos aquí surgirían como consecuencia del reajuste del tamaño de la red neuronal.

4.4.2.2.- Entrenamiento de la Red Neuronal

Sin duda alguna, esta parte es la más importante en la Red Neuronal. Ya se ha comentado que para realizar el entrenamiento se necesitaba ejecutar el mismo comando en todas las estaciones de cálculo. Esta funcionalidad implicaba las siguientes tareas:

I. Elegir los comandos a ejecutar.
II. Enviar los comandos a las máquinas y modificar su comportamiento para responder correctamente
III. Recibir las respuestas de todas las máquinas y entrenar la red.
IV. Fijar un criterio de parada.
V. Establecer un sistema de recuerdo.

I. Elegir los comandos a ejecutar:

Los comandos a ejecutar debían corresponder a procesos que cargaran el sistema tanto en CPU como en memoria y carga de Entrada/Salida. De esta forma, las características dinámicas de las estaciones de cálculo fluctuarían durante el entrenamiento y el aprendizaje sería más completo.

Dichos comando podían variar de unos sistemas a otros, por lo que se ha dejado la tarea de definirlos a decisión del administrador. Aún así se configuraron varios procesos por defecto que cargaran los sistemas en sus distintas características. Ésta configuración se realiza a través de un fichero de configuración donde se indican los comandos a ejecutar.

El envío de los comandos se realiza entre periodos de tiempo aleatorios, cuyo rango puede ser especificado (constante ESPERA_MAX_TRAINING)

II. Enviar los comandos a las máquinas y modificar su comportamiento para responder correctamente.

En este punto hay que resaltar que cuando la aplicación básica estuvo terminada, se comprobó experimentalmente que al enviar el mismo comando a todas las maquinas, sus características dinámicas variaban a la par. **Aunque las máquinas tuvieran características estáticas distintas** (velocidad de CPU, cantidad de memoria,...) **la carga a las que se sometía era la misma** y a la red siempre se le entrenaba con todas las mismas máquinas en un porcentaje de carga similar. Esta situación evitaba, por ejemplo, que la red aprendiera que si una máquina muy potente estaba muy cargada habría de enviar los procesos a otra menos ocupada aunque fuera menos rápida. Esto era debido a que en los ejemplos de entrenamiento nunca se había dado el caso de que una estuviera cargada y la otra no. Para evitarlo, se decidió implementar un **mecanismo que enviara aleatoriamente procesos a alguna estación de cálculo sin que ésta tuviera que contestar de su tiempo de ejecución**. Este mecanismo permite cargar las estaciones de forma asíncrona y la

aleatoriedad en mandar los procesos es **configurable** a través de la constante ENTROPIA.

La primera modificación sufrida por él código de las estaciones de cálculo consistió en adaptarlas para poder ejecutar los comandos de entrenamiento sin que se contactara con ningún cliente. Dicha funcionalidad consistía en diferenciar éstos comandos de los normales y ejecutarlos sin mostrar su salida por pantalla. Acto seguido se enviaría su tiempo de ejecución al módulo de entrenamiento. Aquí también hubo que añadir la modificación que evitara enviar el resultado de los comandos que fueran enviados simplemente como carga.

III. Recibir las respuestas de todas las máquinas y entrenar la red.

Un detalle de implementación a tener muy en cuenta era **que las respuestas de los comandos de entrenamiento podían llegar en cualquier orden y momento**. Esto obligaba a crear una estructura de datos que fuera guardando los resultados de las máquinas hasta que estuvieran todos, y una vez recogidos todos los tiempos devolverlos para ser utilizados en el entrenamiento.

Por ello, una vez lanzado desde el módulo de entrenamiento un comando a las estaciones, se crea una lista con todas las máquinas que debieran contestar (acompañadas por sus características dinámicas) y **dicha lista se identifica con el identificador del proceso (PID)**. Esta estructura, denominada nodo de entrenamiento, se guarda a su vez en otra lista (doblemente enlazada, para su eficaz utilización) donde poder buscar cuando lleguen las respuestas.

Es responsabilidad del AIEngine el enviar los datos dinámicos al módulo de entrenamiento. Esta acción debe realizarse cada vez que llegue la información desde las estaciones hasta el dbm, el cual a su vez, lo transmitirá al AIEngine.

Cuando una respuesta llegue desde una estación de cálculo irá identificada por el PID del proceso y contendrá el tiempo de ejecución del mismo. Se procederá a buscar en la lista el nodo de entrenamiento correspondiente a ese PID y se insertará el tiempo de entrenamiento en la lista con las máquinas. Acto seguido se comprueba si han llegado todas las respuestas, y si se cumple se envía la información con los tiempos para el entrenamiento y se borra el nodo de la lista.

El entrenamiento consiste simplemente en utilizar los datos dinámicos que se guardaron en el nodo de entrenamiento como vector entrada y los tiempos normalizados como vector solución. Esta acción se realiza a través del método *EntrenarRedNeuronal()*, el cual devuelve el error de la red.

La normalización de los tiempos de ejecución consiste en reducir su rango al [0..1], pero invirtiendo los valores. Esto quiere decir que el tiempo más pequeño corresponderá al 1 y el tiempo mayor al 0. Esta operación consiste únicamente en aplicar una fórmula aritmética.

IV. Fijar un criterio de parada.

Esta parte es importante en cualquier implementación de redes neuronales, por lo que se ha intentado hacerla lo más configurable posible.

Como se ha comentado en el apartado anterior, el método llamado al entrenar devuelve el error de la red para un determinado ejemplo. En condiciones normales dicho error irá descendiendo hacia el 0 y hay **que fijar una condición de parada para que no descienda indefinidamente**. Ello se debe a que cuanto más decrezca, irá decrementando en decimales cada vez más pequeños sin llegar nunca hasta el 0 absoluto. Para evitarlo se fija un **rango de parada** que el siguiente error ha de superar respecto al anterior para no detener el entrenamiento. Dicho rango se define en SENSIBILIDAD_PARADA.

Obviamente puede darse que la siguiente ejecución del entrenamiento no supere ese límite, pero que los sucesivos sí que lo hagan. Por ello **se define un margen para esperar a que se supere el error** (indicado en MARGEN_TRAINING).

De este modo surge el concepto de **error mínimo**. El error mínimo se inicializa a 1 (error máximo posible) y si los siguientes entrenamientos diminuyen dicho error mínimo en una cantidad mayor a la definida en SENSIBILIDAD_PARADA, se actualiza el error mínimo con el error del entrenamiento actual. Si los siguientes entrenamientos no consiguen disminuir su valor en MARGEN_TRAINING intentos, se detiene el entrenamiento.

V. Establecer un sistema de recuerdo.

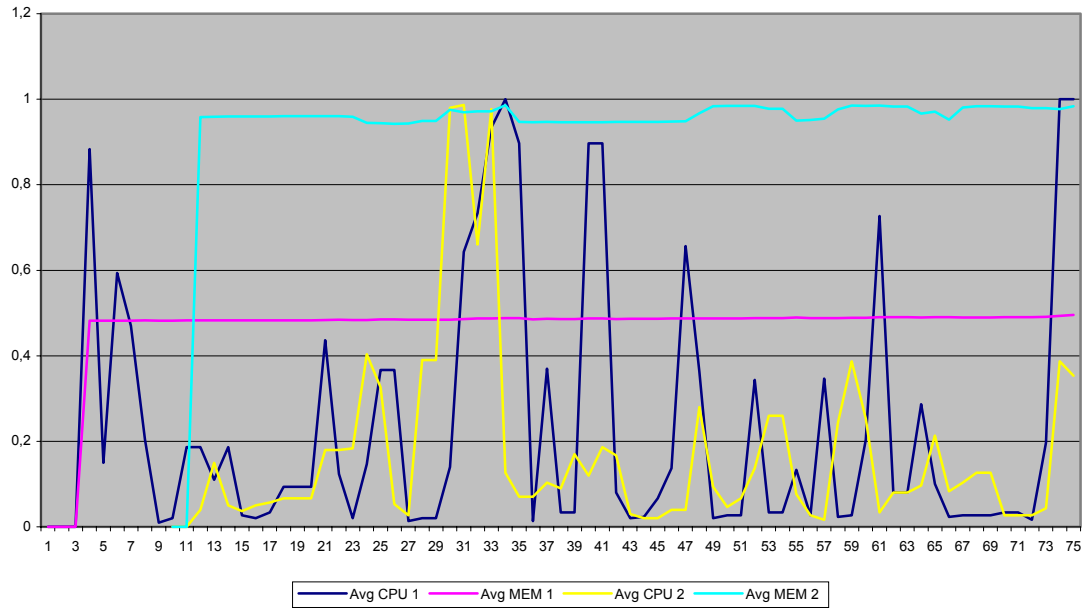
Una vez que la red está entrenada, el proceso encargado de su entrenamiento se detiene. Es de esperar que las características de las máquinas puedan variar con el tiempo debido a que su utilización por parte de los usuarios difiera de la tenida en cuenta durante el entrenamiento, por lo **que pasado un determinado intervalo de tiempo** (indicado por TIEMPO_INVERNAR) **es recomendable despertar el proceso de entrenamiento para que actualice los pesos de la red**. El error mínimo se mantiene respecto al entrenamiento anterior, por lo que no es necesario reiniciar el entrenamiento completamente.

Sin embargo**, hay situaciones en las que sí es necesario comenzar totalmente el entrenamiento**. Este escenario se plantea cuando se añaden o borran estaciones de cálculo al cluster, ya que la red ha de rehacerse de nuevo. En este caso el error mínimo volverá a valer 1 y todo el proceso ha de comenzar de nuevo. (Si el proceso estuviera “invernando”, se despertaría).

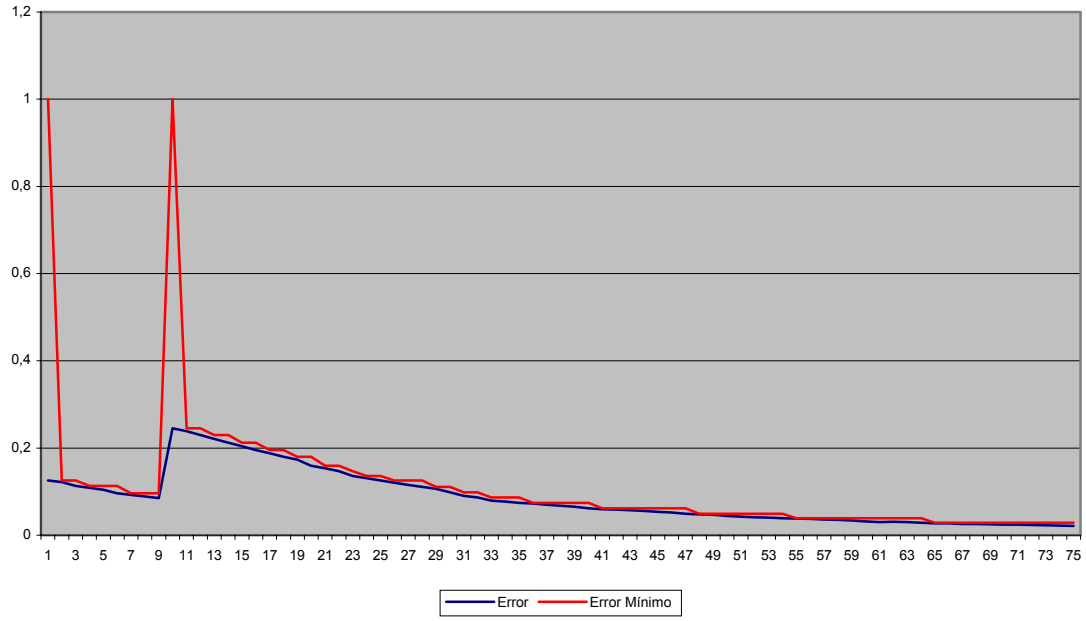
Sólo queda reseñar que todas las constantes identificadas durante esta explicación pueden ser modificadas dentro del archivo configuración *nnetTraining_config.xml*

4.4.2.3.- Ejemplo práctico de entrenamiento.

Para intentar mostrar algunas de las consideraciones teóricas anteriormente expuestas, nos ayudamos de datos obtenidos en un entrenamiento de la red neuronal. En este ejemplo, y por simplicidad, se utilizaron sólo dos estaciones de cálculo con distintas características.



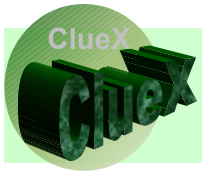
En esta primera gráfica se puede observar que durante el entrenamiento la carga sometida a ambas máquinas es similar. Ésto se manifiesta en el intervalo 29-35. Esta situación podría corromper el aprendizaje por lo que ya se señaló la solución adoptada, consistente en enviar comandos aleatoriamente como se puede comprobar en el intervalo 39-43 donde se muestra una carga desigual.



En la segunda gráfica se muestra la evolución del error mínimo respecto al error devuelto por el entrenamiento.

Primeramente destacar que el pico en el instante 10 se debe a que inicialmente el entrenamiento comenzó con una única máquina y que en ese momento se añadió la segunda. Como era de esperar el error mínimo se reinició a 1 y volvió a comenzar el entrenamiento.

Se puede comprobar que según nos acercamos al 0 el error disminuye a menor ritmo por lo que el escalonamiento del error mínimo se hace más llano debido a que es más difícil supera el valor de SENSIBILIDAD_PARADA. Cuando el error mínimo se mantiene constante durante más iteraciones que las indicadas en MARGEN_TRAINING se finaliza el aprendizaje.



4.5.- Implementación del balanceo de carga.

4.5.1.- Introducción.

Dentro de las posibilidades de implementación de un algoritmo de planificación de procesos, nos queda dejar que sea **el propio administrador** de ClueX el que fije su criterio. Esto es lo que permite nuestra implementación de balanceo de carga.

Concretamente, el administrador **define sus preferencias** acerca de los criterios que deba utilizar el planificador para reartir tareas en un fichero de configuración, determinando la importancia de los diferentes criterios mediante pesos.

A continuación, cada una de las estaciones de cálculo es "evaluada" de forma global, con una simple **fórmula de ponderación** del valor de sus características, en relación con la importancia definida por el administrador. La máquina que consiga la evaluación más alta será la que ejecute el proceso.

Se trata de un método que, aunque es sencillo, no deja de ser importante, ya que pueden existir casos en los que sea necesario acotar la planificación de procesos en nuestro cluster a un subconjunto determinado de procesos.

4.5.2.- El archivo de configuración: *QLBConfig.conf.*

Para que el uso de este fichero resulte sencillo, se ha intentado que su sintáxis sea similar a la de XML. Se utilliza una serie de marcas que delimitan cada una de las características configurables.

El usuario sólo deberá definir mediante un número entero su grado de preferencia por cada característica. La diferencia proporcional de los valores definidos constituirá la configuración del balanceo de carga para las siguientes ejecuciones.

Se ha diseñado una pequeña *jerarquía* de características en el fichero de configuración. Existen 4 características principales y las demás son dependientes de éstas. Tenemos, por tanto 4 grupos de características:

1.- GRUPO CPU.
- Mhz.
- PeakMips.
- Csize.
- AverageCPU.

2.- GRUPO MEM.
- RamSize
- RamSpeed
- AverageMem
- SwapSize
- AverageSwap
- HDSpeed

SysBusSpeedMhz.

3.- GRUPO SWAP.
SwapSize.
AverageSwap.
HDSpeed.

4.- GRUPO IO.
HDSize.
HDSpeed.
IOBusSpeed.

Gracias a esta estructuración de las características en grupos, logramos que, **modificando la importancia de cualquiera de las principales, automáticamente lo estemos haciendo sobre todas aquellas que figuran en su grupo**. Por ejemplo, si el usuario sube el grado de importancia de *CPU*, automáticamente lo estará haciendo sobre las características *Mhz, PeakMips, Csize, y AverageCPU.* En el caso en que alguna de las propiedades hijas tenga definido un valor concreto, éste prevalecerá por encima del valor definido para su padre.

4.5.3.- Implementación.

Lo primero que debe realizar el algoritmo es extraer del fichero de configuración los pesos de las características, para poder aplicarlos posteriormente en la ponderación de cada estación de cálculo. Se ha logrado implementar este "parsing" del fichero de forma que incluso el usuario pueda modificar el fichero *QLBConfig.conf* en caliente; es decir, sin tener que volver a arrancar ningún proceso de ClueX. El procedimiento que realiza el citado "parsing" se llama *parseConfigFile*, y guarda el valor de los pesos en la variable *headerQLB.*

Posteriormente, cada una de las máquinas es evaluada, con la función *valorQLB*, y consultando los valores de *headerQLB.*

Como ejemplo incluimos la DTD y un fichero XML de configuración:

```
<!ELEMENT CluexQLBConfig ( CPU, MEM, SWAP, IO, Mhz?, PeakMips?, AverageCpu?, RamSize?,
RamSpeed?, CSize?, AverageMem?, SwapSize?, AverageSwap?,IOBusSpeed?, ysBusSpeed?, HDSize?,
HDSpeed?)>
<!ELEMENT CPU (#PCDATA)>
<!ELEMENT MEM (#PCDATA)>
<!ELEMENT SWAP (#PCDATA)>
<!ELEMENT IO (#PCDATA)>
<!ELEMENT IOBusSpeed (#PCDATA)>
<!ELEMENT SysBusSpeed (#PCDATA)>
<!ELEMENT Mhz (#PCDATA)>
<!ELEMENT CSize (#PCDATA)>
<!ELEMENT PeakMips (#PCDATA)>
<!ELEMENT RamSize (#PCDATA)>
<!ELEMENT RamSpeed (#PCDATA)>
```

```
<!ELEMENT SwapSize (#PCDATA)>
<!ELEMENT HDSize (#PCDATA)>
<!ELEMENT HDSpeed (#PCDATA)>
<!ELEMENT AverageCpu (#PCDATA)>
<!ELEMENT AverageMem (#PCDATA)>
<!ELEMENT AverageSwap (#PCDATA)>



<CluexQLBConfig>
<!-- QLBconfig.conf ClueX Qualified Load-Balancing configuration file -->
<!-- With this file you can tailor manually the mechanism of ClueX Load Balancing-->
<!-- You have 4 general parameters to set weights to sets of parameters, by example-->
<!-- CPU parameter gives automatic weights to CPU-related machine parameters-->
<!-- You can give a value to CPU and later overwrite this values with more -->
<!-- specific parameters, setting for example a general weight of 1 and-->
<!-- a particular weight of 2 to CPU load to increase the load balancing towards-->
<!-- those machines with less CPU cycles busy-->
<!---->
<!-- Be careful, some global parameters overlap values, like MEM and SWAP-->

<!-- Global parameters-->
<!-- CPU includes Mhz, PeakMips, CSize and AverageCPU -->
        <CPU>1</CPU>
<!-- MEM includes RamSize, RamSpeed, AverageMem, SwapSize, AverageSwap, HDSpeed,
SysBusSpeed-->
        <MEM>0</MEM>
<!-- SWAP includes SwapSize, AverageSwap, HDSpeed-->
        <SWAP>0</SWAP>
<!-- IO includes HDSize, HDSpeed, IOBusSpeed-->
        <IO>0</IO>
<!-- Specific parameters. If you don't want to overwrite global parameters, comment out-->
<!-- these lines-->
<!-- Mhz ==> CPU Mhz-->
        <Mhz>0</Mhz>
<!-- PeakMips ==> Maximum millions of integer instructions per second.-->
        <PeakMips>0</PeakMips>
<!-- AverageCPU ==> CPU load percentage-->
        <AverageCPU>3</AverageCPU>
<!-- RamSize ==> Guess?-->
        <RamSize>0</RamSize>
<!-- RamSpeed ==> RAM Mhz-->
        <RamSpeed>0</RamSpeed>
<!-- CSize ==> 2nd level Cache Size-->
        <CSize>0</CSize>
<!-- AverageMem ==> Percent of free RAM-->
        <AverageMem>0</AverageMem>
<!-- Size of Swap partition-->
        <SwapSize>0</SwapSize>
<!-- AverageSwap ==> Percent of free swap-->
        <AverageSwap>0</AverageSwap>
<!-- IOBusSpeed ==> Speed of IO Bus-->
        <IOBusSpeed>0</IOBusSpeed>
<!-- SysBusSpeed ==> Speed of CPU-MEM bus-->
        <SysBusSpeed>0</SysBusSpeed>
<!-- HDSize ==> Size of Hard Disks-->
        <HDSize>1</HDSize>
<!-- HDSpeed ==> Speed of Hard Disk Device-->
        <HDSpeed>0</HDSpeed>
</CluexQLBConfig>
```

MÓDULO V

SEGURIDAD:

CRIPTOGRAFÍA

IDENTIFICACIÓN

5.- Seguridad: Criptografía e Identificación.

5.1.- Introducción.

Un aspecto muy importante en cualquier aplicación de este tipo es la **seguridad en las transmisiones**. En Cluex se ha tenido en cuenta este requisito y se ha desarrollado una librería que permite a los distintos módulos intercomunicarse de forma segura. Aparte se ha definido un **sistema de identificación** tanto para los clientes como para las estaciones de cálculo

Entre los distintos sistemas criptrográficos existentes, podemos destacar dos familias: **sistemas simétricos y asimétricos**.

Los primeros utilizan una clave común para cifrar y descrifrar los datos que ha de ser conocida por ambos extremos de la comunicación. Normalmente este tipo de algoritmos son los de ejecución más rápida y por ello son utilizados en la encriptación de transmisiones. Por otro lado, el principal inconveniente de estos sistemas es la necesidad del intercambio de la contraseña entre el emisor y receptor de la comunicación. Como ejemplos de este tipo de algoritmos tenemos el ya clásico **DES**, **triple-DES** y su sucesor: el **AES**.

Los **sistemas asimétricos** permiten el intercambio de información sin la necesidad del conocimiento común de la clave. En estos algoritmos el receptor genera dos claves: una **pública** y otra **privada**. La clave pública permite que el emisor encripte los datos de forma que sólo el receptor que posea la clave privada correspondiente pueda descifrarlos. Sin duda el algoritmo de este tipo más extendido es el **RSA**.

Dentro de Cluex se necesitaba tanto la transmisión encriptada de los datos como el intercambio seguro de la clave para poder realizarla. Por ello se decidió utilizar los algoritmos **AES y RSA**.

5.2.- Algoritmos criptográficos.

Para poder utilizar la criptografía se adaptaron dos implementaciones, tanto de AES como de RSA y se incluyó todo el código en la librería *libCCrypt.a*.

5.2.1.- RSA.

La implementación de RSA está escrita **en C++** y propone **dos clases**: una encargada de implementar enteros de longitud arbitraria (necesarios para el algoritmo) y otra clase que implementaba el propio algoritmo.

Como principal extensión a esta implementación se añadieron dos métodos encargados de "**serializar**" a un buffer de bytes la clave pública para poder ser enviada a través de nuestro módulo de red.

A continuación se desarrolló el código encargado de hacer de interfaz entre esa implementación y nuestro código (rsa2c.h). En este archivo se incluyen métodos para

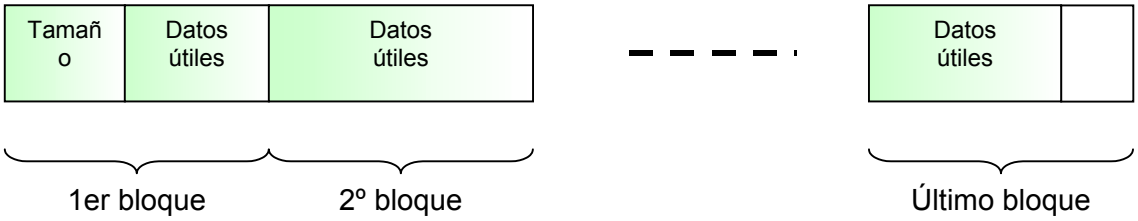
la **creación de las claves de forma aleatoria, su intercambio** y como no podría ser de otra manera su **encriptado y desencriptado**.

5.2.2.- AES.

Al igual que con el código del RSA, para el AES (también conocido como Rijndael) se desarrollo código que hiciera de interfaz con Cluex (aes2c.h).

En este código se incluyeron métodos para inicializar el algoritmo con la clave simétrica y el encriptado y desencriptado.

Un característica central de este algoritmo es que el **cifrado se realiza por bloques de 16 bytes**, por lo que **hubo que definir un "estándar" que nos permitiera cifrar estructuras de cualquier tamaño**. El sistema utilizado consiste en reservar los primeros 4 bytes del primer bloque para indicar la longitud total de los datos útiles encriptados y así poder descartar el sobrante del último bloque:

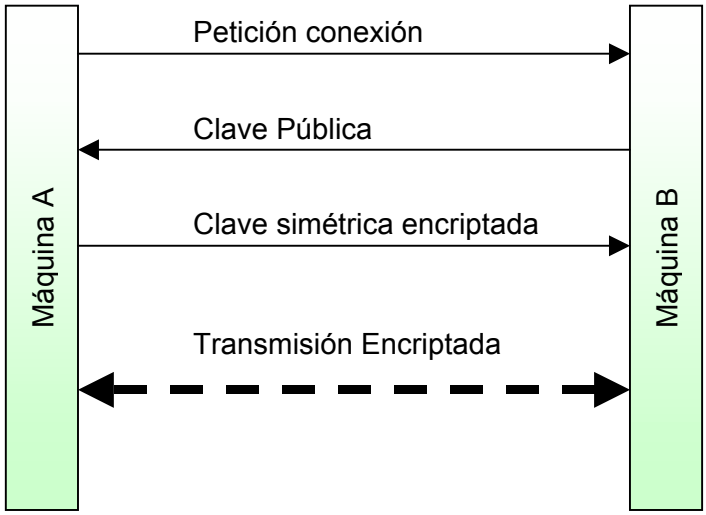


Como consecuencia de este formato se desarrollaron también **métodos encargados de averiguar de cuantos bloques está formado un buffer encriptado** (a partir sólo del primer bloque).

5.3.- Protocolo de comunicación encriptada.

La utilización de la criptografía en todas las comunicaciones dentro de ClueX sigue el mismo esquema:

1. **Petición de Conexión** desde la máquina A hasta la máquina B.
2. La máquina **B envía su clave pública (RSA) hasta A**.
3. **A encripta su clave simétrica** (para AES) **con la clave pública de B y se la envía**.
4. **B** recibe la clave simétrica de A y **la desencripta usando su clave privada**.
5. Una vez intercambiada la clave simétrica el resto de transacciones se realiza **encriptado con AES**.



Una vez realizado este intercambio de claves, el resto de las comunicaciones sigue su propio protocolo, según lo indicado en los apartados de comunicaciones y arquitectura.

5.4.- Identificación.

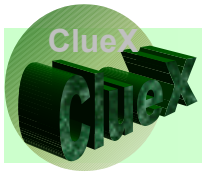
Aunque las comunicaciones se realicen de forma totalmente segura, ningún sistema criptográfico tiene validez si no se puede asegurar que la máquina con la que nos estemos comunicando sea una máquina segura.

Por ello, se han desarrollado dos sistemas de identificación: Una para los usuarios, consistente en **contraseña** y otro para las estaciones de cálculo consistente en la **dirección de red**.

Para la identificación de los usuarios, el administrador de Cluex puede definir **nombres de usuario** (login) y sus correspondientes **contraseñas** (passwords). Esta información se almacena en el fichero XML passwd.cluex.

```
DTD:
<!ELEMENT ClueXUsers (ClueXUser*)>
<!ELEMENT ClueXUser (UserName, PassWord)>
<!ELEMENT UserName (#PCDATA)>
<!ELEMENT PassWord (#PCDATA)>


Ejemplo:

<ClueXUsers>
<ClueXUser><UserName>user1</UserName><PassWord>user1</PassWord></ClueXUser>
```

```
<ClueXUser><UserName>user2</UserName><PassWord>user2</PassWord></ClueXUser>
<ClueXUser><UserName>user3</UserName><PassWord>user3</PassWord></ClueXUser>
```




```
<ClueXUser><UserName>admin</UserName><PassWord>admin</PassWord></ClueXUser>
</ClueXUsers>
```

Respecto a la identificación de las estaciones de cálculo, era impensable el que cada máquina tuviera su propia contraseña y que el administrador se tuviera que encargar de introducirla cada vez que se añadiera una máquina. En vez de ello el sistema utilizado **se basa en la dirección IP de la máquina.**

El administrador puede definir qué máquinas están admitidas en Cluex a través de una dirección de red y una máscara de subred. Estos 2 datos permiten de una forma muy flexible identificar a las estaciones de cálculo.

La definición de las máquinas admitidas se realiza en el fichero XML *hosts.cluex*.

DTD:
<!ELEMENT ClueXHosts (CluexHost*)>
<!ELEMENT CluexHost (Net, NetMask)>
<!ELEMENT Net (#PCDATA)>
<!ELEMENT NetMask (#PCDATA)>

Ejemplo (admitimos todas las máquinas):

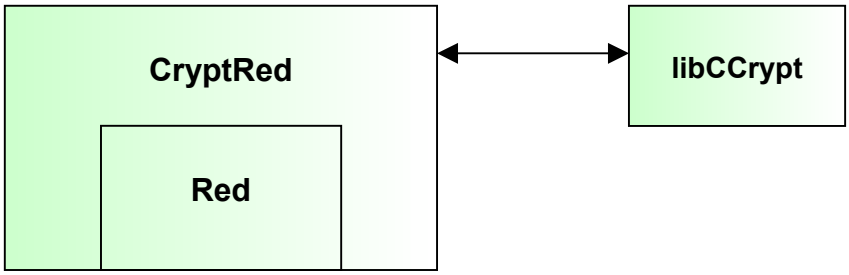
<ClueXHosts>
<ClueXHost><Net>0.0.0.0</Net><NetMask>0.0.0.0</NetMask></ClueXHost>
</ClueXHosts>

5.5.- Integración en ClueX.

Una vez creados ambos sistemas criptográficos se desarrolló un **módulo encargado de realizar las transmisiones encriptadas** con la misma transparencia que el módulo de red.

Si recordamos, el módulo de red ofrecía primitivas para en envío y recepción de datos tanto a través de TCP como UDP. De esta forma el nuevo módulo debía ofrecer las mismas primitivas y ocultar a su vez todos los detalles del sistema utilizado. Debía extraer la información de los bloques y devolverla evitando incluir tanto los primeros 4 bytes como el espacio sobrante, así como calcular y recibir el número de bloques necesarios para un determinado buffer de datos.

El módulo desarrollado (*cryptRed*) se coloca justamente por encima del módulo de red ya que utiliza sus métodos para transmitir los datos encriptados, y utiliza la librería de criptografía:



De esta forma la utilización de las transmisiones encriptadas es **totalmente transparente** al resto de módulos.

Respecto a la identificación, la identificación de los usuarios se realiza en el módulo sched_recepcion. Cuando un cliente intenta conectar, se le pide su login y password y se coteja con los contenidos en el fichero passwd.cluex. Si los datos no son válidos se cierra la conexión.

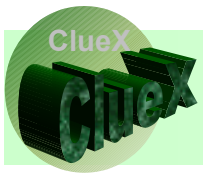
Para las estaciones de cálculo, se obtiene la dirección IP de la máquina solicitante. A dicha dirección se le aplica la operación AND con las máscaras de subred y se comprueba si la dirección resultante coincide con la correspondiente dirección de red.

El módulo encargado de realizar todo esto es AgentServer, ya que es el primer proceso del planificador con el que se comunican las estaciones de cálculo.

MÓDULO VI

LA INTERFAZ

GRÁFICA



6.- La interfaz gráfica.

6.1.- Introducción.

Se ha considerado muy interesante la posibilidad de **manejo y gestión no centralizados de ClueX**, y es lo que se ha pretendido con la implementación de la Interfaz Gráfica (de aquí en adelante, GUI).

La GUI está implementada **en JAVA**, y puede ser ejecutada desde cualquier máquina del sistema. Es capaz de comunicarse con el planificador, y de mostrar prácticamente todos los datos disponibles para una gestión adecuada del clúster. Permite una **flexibilidad y comodidad en principio no imaginables**.

Además, garantizamos la seguridad del sistema, gracias a la **encriptación de las comunicaciones** que utiliza la GUI.

6.2.- Implementación.

6.2.1.- Las comunicaciones.

El método de comunicación de la GUI con cualquiera de las máquinas del sistema está basado en la **construcción de mensajes XML**. Se distinguió entre los posibles tipos de mensaje que resultaran necesarios, y se diseñó para cada uno de ellos una sintáxis en DTD que albergase toda la información necesaria para la comunicación. Dicha sintáxis se encuentra debidamente especificada en los archivos *query.dtd* y *answer.dtd*.

Los tipos de mensaje que se han distinguido son:

De **consulta** desde la GUI al planificador, o de **respuesta** en el sentido contrario.

Las consultas realizadas por la GUI pueden tener como finalidad el obtener información desde el planificador (“Query”) o realizar un cambio en su configuración:

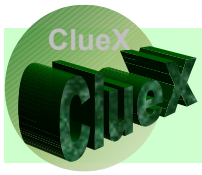
```
<!ELEMENT Cluex (Query | Change)>
```

Cuando la consulta pretende obtener información de una máquina, el mensaje debe **especificar la máquina objeto** de la consulta de qué naturaleza es la información que solicitamos:

```
<!ELEMENT Query (QueryType, Ip?)>
<!ELEMENT QueryType (GlobalState | Client | Scheduler | CalcStation | HistoryList)>
```

Los mensajes “Change” tienen como finalidad cambiar algún parámetro de configuración del planficador, acerca de la política, la autentificación,

la configuración del algoritmo del balanceo de carga, o la de la red neuronal:




```
<!ELEMENT Change (Policy | Auth | CluexQLBConfig | NNetInfo)>
```

A su vez, cada uno de esos tipos de mensaje contienen toda la información que necesitan para llevar a cabo la función que deben desempeñar.

Por otra parte, las respuestas son generadas por los correspondientes procesos encargados de recibir las consultas. Existen varios tipos de respuestas: Lista de máquinas, Estado de una máquina, Historial y Procesos en ejecución:

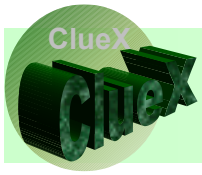
```
<!ELEMENT Answer (ListAnswer | StateAnswer | HistoryList | ProcessAnswer)>
<!ELEMENT ListAnswer (Ip*)>
<!ATTLIST ListAnswer
        type (client | scheduler | calcstation) #REQUIRED
```

Se ha llegado a refinar la sintáxis de tal manera que, como vemos más adelante, es posible visualizar toda la información que el sistema puede proporcionar, sin ninguna excepción.

La definición completa del archivo *query.dtd* es:

```
<!ELEMENT Cluex (Query | Change)>
<!ELEMENT Query (QueryType, Ip?)>

<!ELEMENT QueryType (GlobalState | Client | Scheduler | CalcStation | HistoryList)>
<!ELEMENT GlobalState EMPTY>
<!ELEMENT Client EMPTY>
<!ELEMENT Scheduler EMPTY>
<!ELEMENT CalcStation EMPTY>
<!ELEMENT HistoryList EMPTY>

<!ELEMENT Ip (#PCDATA)>

<!ELEMENT Change (Policy | Auth | CluexQLBConfig | NNetInfo)>

<!ELEMENT Policy (#PCDATA)>

<!ELEMENT Auth (CluexUsers,CluexHost)>

<!ELEMENT ClueXUsers (ClueXUser*)>
<!ELEMENT ClueXUser (UserName, PassWord)>
<!ELEMENT UserName (#PCDATA)>
<!ELEMENT PassWord (#PCDATA)>

<!ELEMENT ClueXHosts (CluexHost*)>
<!ELEMENT CluexHost (Net, NetMask)>
<!ELEMENT Net (#PCDATA)>
```

<!ELEMENT NetMask (#PCDATA)
<!ELEMENT CluexQLBConfig (CPU, MEM, SWAP, IO, Mhz?, PeakMips?, AverageCpu?, RamSize?, RamSpeed?, CSize?, AverageMem?, SwapSize?, AverageSwap?,IOBusSpeed?, ysBusSpeed?, HDSize?, HDSpeed?)>



6.- La interfaz gráfica.

<!ELEMENT CPU (#PCDATA)>
<!ELEMENT MEM (#PCDATA)>
<!ELEMENT SWAP (#PCDATA)>
<!ELEMENT IO (#PCDATA)>
<!ELEMENT IOBusSpeed (#PCDATA)>
<!ELEMENT SysBusSpeed (#PCDATA)>
<!ELEMENT Mhz (#PCDATA)>
<!ELEMENT CSize (#PCDATA)>
<!ELEMENT PeakMips (#PCDATA)>
<!ELEMENT RamSize (#PCDATA)>
<!ELEMENT RamSpeed (#PCDATA)>
<!ELEMENT SwapSize (#PCDATA)>
<!ELEMENT HDSize (#PCDATA)>
<!ELEMENT HDSpeed (#PCDATA)>
<!ELEMENT AverageCpu (#PCDATA)>
<!ELEMENT AverageMem (#PCDATA)>
<!ELEMENT AverageSwap (#PCDATA)>

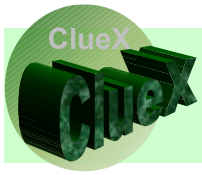
<!ELEMENT NNetInfo (NNetConfig, NNetTraining, NNetTrainingCommands)>
<!ELEMENT NNetConfig (Tasa, Momento)>
<!ELEMENT Tasa (#PCDATA)>
<!ELEMENT Momento (#PCDATA)>
<!ELEMENT NNetTraining (Espera, Entropia, Margen, Sensibilidad, Invernar)>
<!ELEMENT Espera (#PCDATA)>
<!ELEMENT Entropia (#PCDATA)>
<!ELEMENT Margen (#PCDATA)>
<!ELEMENT Sensibilidad (#PCDATA)>
<!ELEMENT Invernar (#PCDATA)>
<!ELEMENT NNetTrainingCommands (cmd*)>
<!ELEMENT cmd (#PCDATA)>

Sus correspondientes respuestas se definen en answer.dtd (obviamos definiciones repetidas):

<!ELEMENT Cluex (Answer)>
<!ELEMENT Answer (ListAnswer | StateAnswer | HistoryList | ProcessAnswer)>
<!ELEMENT ListAnswer (Ip*)>
<!ATTLIST ListAnswer
type (client | scheduler | calcstation) #REQUIRED
>
<!ELEMENT Ip (#PCDATA)>

<!ELEMENT ProcessAnswer (ExecProcess*)>
<!ELEMENT ExecProcess (Command, GPid, ClientIP, CalcIP)>
<!ELEMENT Command (#PCDATA)>
<!ELEMENT ClientIP (#PCDATA)>
<!ELEMENT CalcIP (#PCDATA)>
<!ELEMENT GPid (#PCDATA)>

<!ELEMENT HistoryList (HistoryProcess*)>
<!ELEMENT HistoryProcess (GPid, Command, Date, Time, ClientIP, CalcIP)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Time (#PCDATA)>



<!ELEMENT StateAnswer (Ip, (StateClient | StateScheduler | StateCalcStation))>

<!ELEMENT StateClient (Ip, Port, Certificate)>
<!ELEMENT Port (#PCDATA)>
<!ELEMENT Certificate (#PCDATA)>

<!ELEMENT StateScheduler (Policy, SchedState, ClueXUsers, ClueXHosts, CluexQLBConfig, NNetInfo)>

<!ELEMENT Policy (#PCDATA)>

<!ELEMENT SchedState (SchedCPU, SchedMEM)>
<!ELEMENT SchedCPU (#PCDATA)>
<!ELEMENT SchedMEM (#PCDATA)>

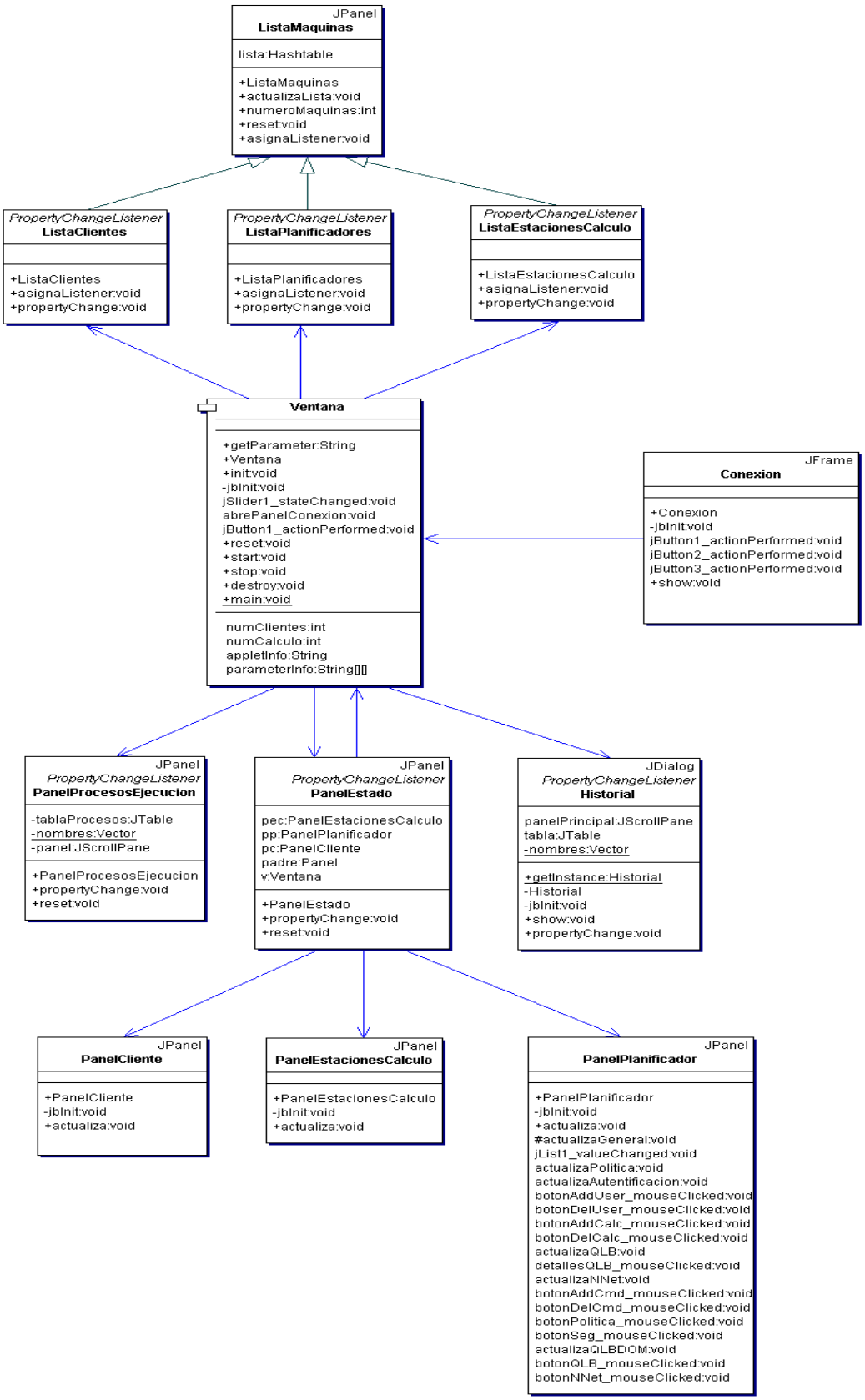
6.2.2.- El diseño de clases.

El diseño de la GUI, al igual que el resto del proyecto, se ha realizado de la forma más clara y extensible posible utilizando **patrones de diseño** según fueran demandados por la funcionalidad de las clases. Su división en paquetes es:

6.2.2.1.- es.ucm.sip.cluexgui.gui:

Define los componentes gráficos utilizados: Ventanas, cuadros de diálogo, paneles, ...

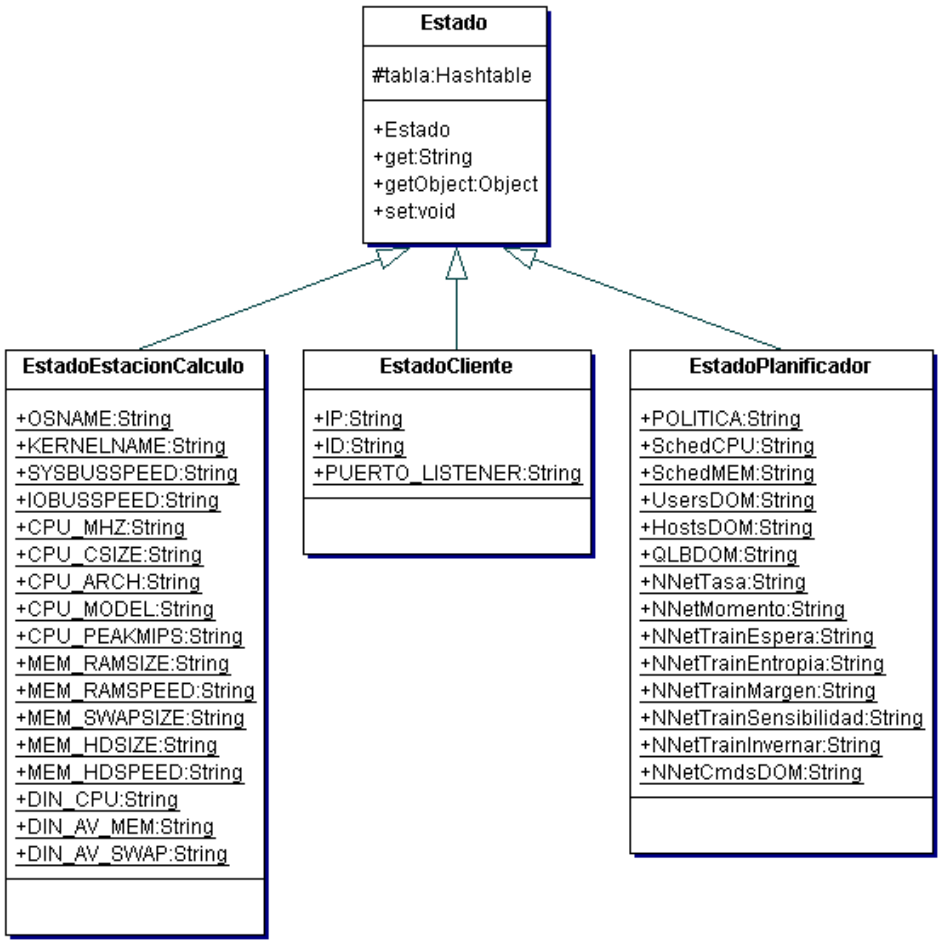
La clase principal y directora de todos los componentes gráficos es la clase *Ventana*. Ésta se compone de tres paneles encargados de mostrar las listas de máquinas, una ventana que permite realizar la conexión y paneles encargados de mostrar la información. El panel informativo más importante es el encargado de mostrar el estado de los tres tipos de máquinas. En este diagrama se define el diseño:


JPanel
ListaMaquinas
lista:Hashtable
+ListaMaquinas
+actualizaLista:void
+numeroMaquinas:int
+reset:void
+asignaListener:void
PropertyChangeListener
ListaClientes
+ListaClientes
+asignaListener:void
+propertyChange:void
PropertyChangeListener
ListaPlanificadores
+ListaPlanificadores
+asignaListener:void
+propertyChange:void
PropertyChangeListener
ListaEstacionesCalculo
+ListaEstacionesCalculo
+asignaListener:void
+propertyChange:void
Ventana
+getParameter:String
+Ventana
+init:void
-jbInit:void
jSlider1_stateChanged:void
abrePanelConexion:void
jButton1_actionPerformed:void
+reset:void
+start:void
+stop:void
+destroy:void
+main:void
numClientes:int
numCalculo:int
appletInfo:String
parameterInfo:String[][]
JFrame
Conexion
+Conexion
-jbInit:void
jButton1_actionPerformed:void
jButton2_actionPerformed:void
jButton3_actionPerformed:void
+show:void
JPanel
PropertyChangeListener
PanelProcesosEjecucion
-tablaProcesos:JTable
-nombres:Vector
-panel:JScrollPane
+PanelProcesosEjecucion
+propertyChange:void
+reset:void
JPanel
PropertyChangeListener
PanelEstado
pec:PanelEstacionesCalculo
pp:PanelPlanificador
pc:PanelCliente
padre:Panel
v:Ventana
+PanelEstado
+propertyChange:void
+reset:void
JDialog
PropertyChangeListener
Historial
panelPrincipal:JScrollPane
tabla:JTable
-nombres:Vector
+getInstance:Historial
-Historial
-jbInit:void
+show:void
+propertyChange:void
JPanel
PanelCliente
+PanelCliente
-jbInit:void
+actualiza:void
JPanel
PanelEstacionesCalculo
+PanelEstacionesCalculo
-jbInit:void
+actualiza:void
JPanel
PanelPlanificador
+PanelPlanificador
-jbInit:void
+actualiza:void
#actualizaGeneral:void
jList1_valueChanged:void
actualizaPolitica:void
actualizaAutentificacion:void
botonAddUser_mouseClicked:void
botonDelUser_mouseClicked:void
botonAddCalc_mouseClicked:void
botonDelCalc_mouseClicked:void
actualizaQLB:void
detallesQLB_mouseClicked:void
actualizaNNet:void
botonAddCmd_mouseClicked:void
botonDelCmd_mouseClicked:void
botonPolitica_mouseClicked:void
botonSeg_mouseClicked:void
actualizaQLBDOM:void
botonQLB_mouseClicked:void
botonNNet_mouseClicked:void

6.2.2.2.- es.ucm.sip.cluexgui.info:

En este paquete se definen las clases utilizadas en la representación de los datos utilizados. Si reflexionamos sobre la información a mostrar sobre los distintos tipos de máquinas, observamos que todos los datos tienen la forma <*propiedad:valor*>, por lo que la estructura más apropiada y eficiente para guardarlos es una **tabla Hash.**

Por ello se define una clase padre que encapsula una tabla de este tipo, y sus subclases encargadas de representar el estado de cada tipo de máquina definirán las propiedades propias de cada una. De esta forma **el diseño es totalmente extensible a nuevas características**:



6.2.2.3.- es.ucm.sip.cluexgui.red:

Sin duda, este es el paquete más importante, ya que **orquesta todo el flujo de comunicación con el planificador.**

Empezamos por la clase *ConexionRed*. Ésta clase se encarga de conectar con el planificador, realizar la autentificación mediante contraseña

e inicializar la encriptación de datos (como ya se verá a continuación). Ofrece a sus clases clientes el **socket** de conexión para la comunicación.

Como debe ser accesible a través de distintos puntos del código se utiliza el patrón **Singleton** para su implementación. Además este patrón nos garantiza la existencia de una única conexión en cada momento.

La autentificación se realiza a través de la contraseña de un usuario especial: *"admin"*. Este usuario ha de estar definido en el fichero de passwords del Cluex. Cuando el planificador inicie una sesión gui, leerá esa contraseña y encriptará los datos con ella. Por su parte la GUI pedirá esa contraseña al usuario y también la utilizará para la encriptación. Si las contraseñas no coinciden, no habrá comunicación.

Aparte tenemos las clases *Petición* y *Respuestas*. Su función es transmitir y recibir los datos. Como ambas clases también han de ser accesibles para varias clases utilizamos también el **patrón Singleton.**

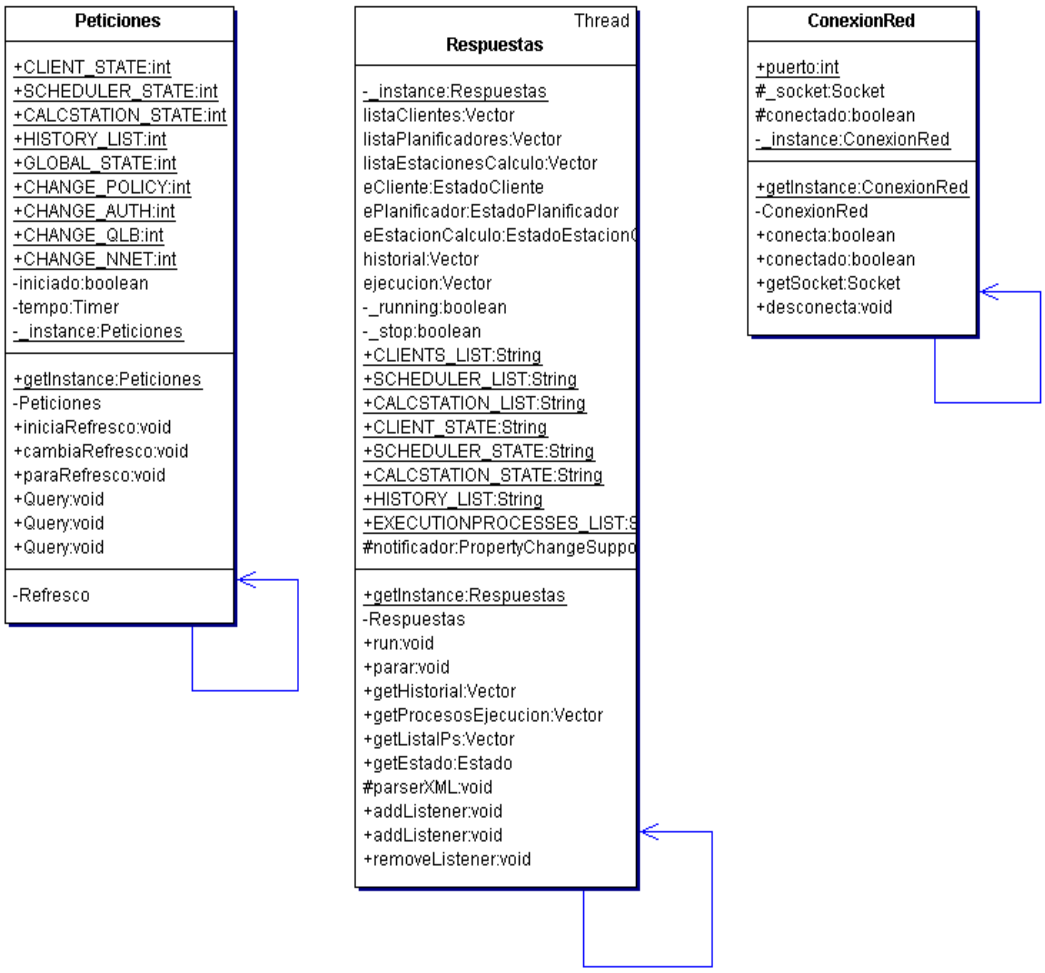
La transmisión de la información hacia el planificador se realiza a través del modelo de eventos, es decir, cuando el usuario requiere la acción. Sin embargo, la recepción de la información puede ocurrir en cualquier instante. No es posible bloquear toda la aplicación a la espera de respuestas, ni preguntar periódicamente por la información venidera. Por ello, la clase *Respuestas* se ejecuta como una **hebra paralela** a la aplicación principal.

Una vez recibidos los datos ha de comunicarlos a los distintos elementos del programa para que sean tratados y mostrados. Una deficiencia muy importante del diseño consistiría en tener que almacenar en ésta clase una referencia a todos los objetos interesados en manejar la información, ya que a priori no se conoce cuantos componentes están a la espera y en una futura extensión de la aplicación habría que rehacer mucho código. Para evitar este problema se implementa en esta clase el patrón **Observer** para que los objetos oyentes sean avisados cuando su información esté disponible.

De esta forma obtenemos un modelo de comunicación libre de bloqueos, en el que los datos serán mostrados al ser recibidos, sin ninguna espera adicional.

Al recibir los datos, un **parser de XML** tratará la información y la almacenará en la correspondiente clase del paquete *info.* A continuación avisará a los componentes interesados en dichos datos.

Como ya perfilamos anteriormente la clase *peticiones* es la **encargada de generar los XML** con la información o peticiones a enviar al planficador y mandar dichos datos a través de la red.



6.2.2.4.- es.ucm.sip.cluexgui.cryto:

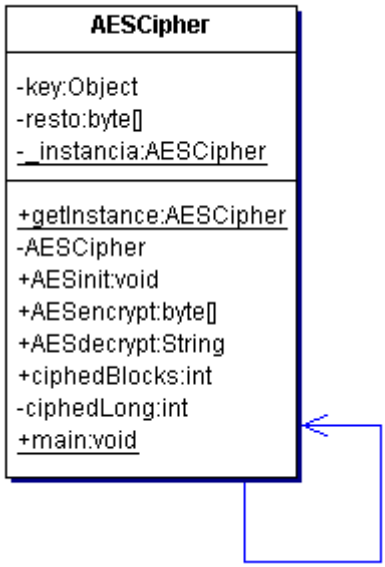
Este paquete es el encargado de **encriptar y desencriptar** los datos transimitidos o recibidos.

En el apartado anterior no se ha comentado que los datos recibidos desde el planificador están encriptados mediante el **algoritmo AES**, y consecuentemente, los datos enviados también tendrán que estarlo.

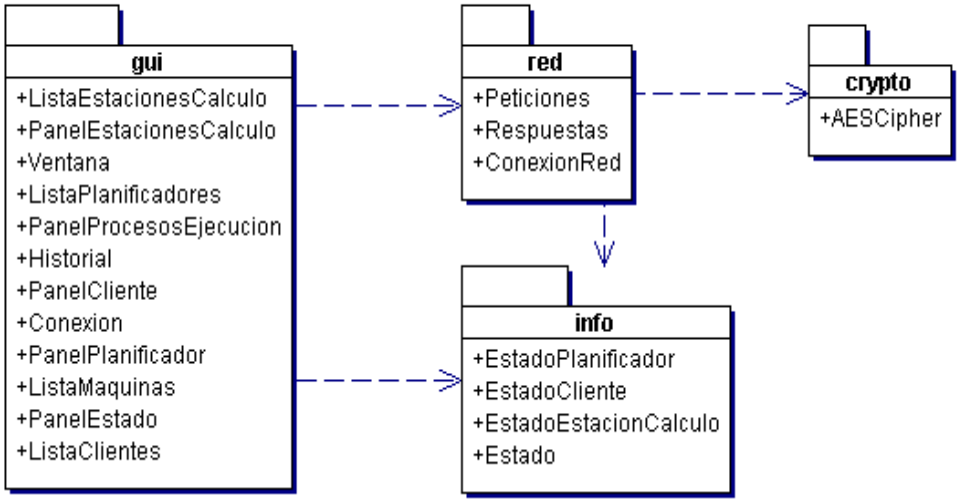
Los datos encriptados que se intercambia con el planificador siguen el mismo esquema que los utilizados para las comunicaciones dentro de Cluex. Esta información se detalla en el capítulo de criptografía, pero en líneas generales se compone de 4 primeros bytes indicando la longitud de los datos a transmitir seguido de dichos datos. Toda la información se encripta en bloques de 16 bytes.

Este paquete se compone de la clase *AESCipher*, encargada de toda la funcionalidad presentada. Ésta incluye tareas como la **adaptación de los bytes de C a Java** (realizando el complemento a 2 cuando es requerido) y el **empaquetado o desempaquetado de los datos** así como su **encriptación y desencriptación**.

En el siguiente diagrama se observan los métodos implementados, que siguen exactamente el mismo esquema que los desarrollados en C para el clúster:



Para finalizar presentamos el **diagrama global de los paquetes**, donde se indican claramente sus relaciones:


gui
+ListaEstacionesCalculo
+PanelEstacionesCalculo
+Ventana
+ListaPlanificadores
+PanelProcesosEjecucion
+Historial
+PanelCliente
+Conexion
+PanelPlanificador
+ListaMaquinas
+PanelEstado
+ListaClientes
red
+Peticiones
+Respuestas
+ConexionRed
crypto
+AESCipher
info
+EstadoPlanificador
+EstadoCliente
+EstadoEstacionCalculo
+Estado

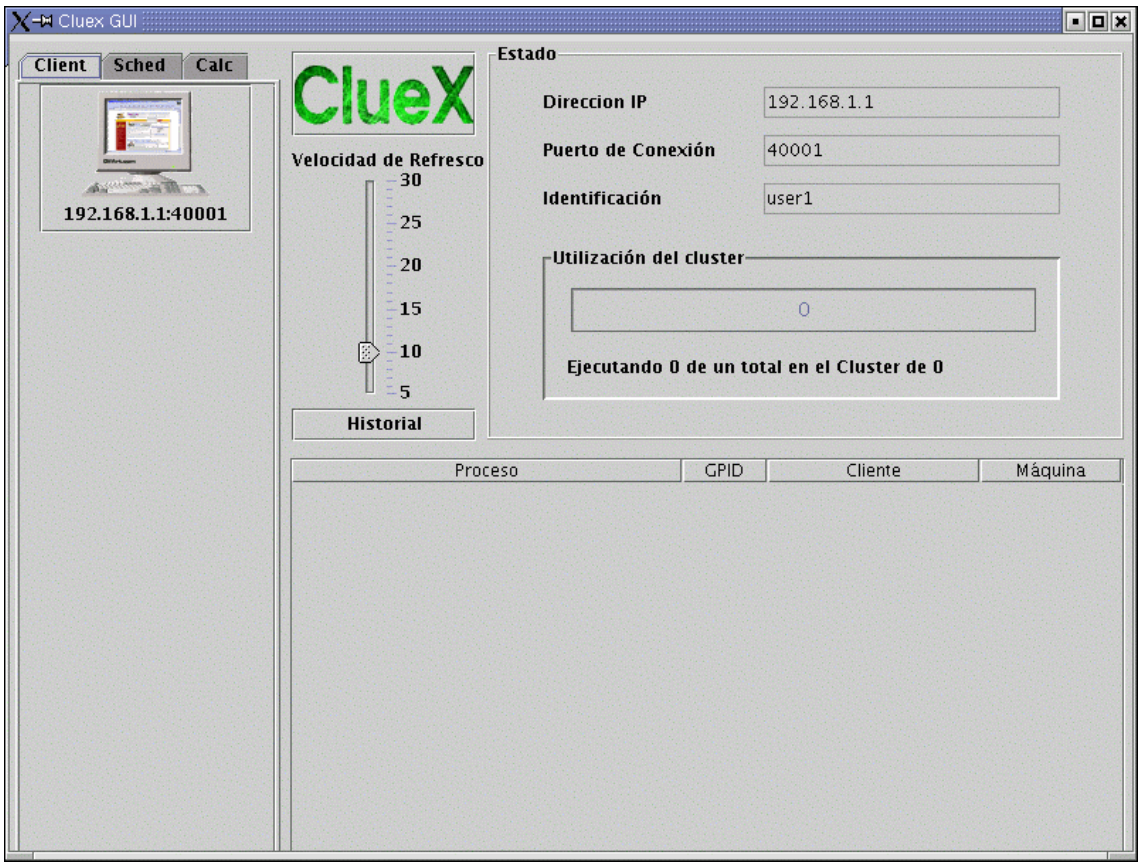
6.3.- Forma de uso de la GUI.

6.3.1.- Información de los clientes

La utilización de la GUI es intuitiva y no presenta dificultades. Lo primero que hay que hacer es **conectarla al planificador**, para lo cual, es necesario introducir su IP y la contraseña del administrador.

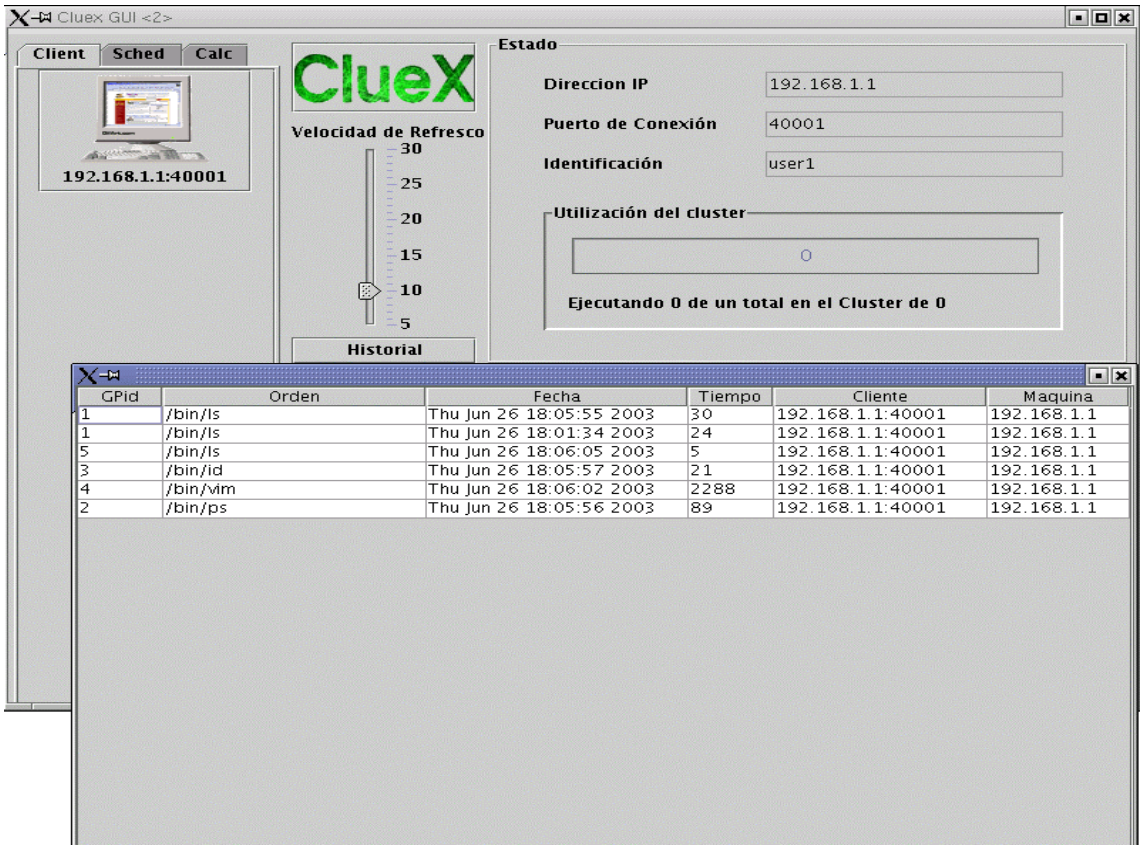
Una vez realizada la conexión, se puede visualizar un icono de cada una de las máquinas dadas de alta en el sistema. Las pestañas de la parte izquierda permiten consultar la información en función de los tipos de máquina de ClueX**. La información es actualizada dinámicamente**, y los cambios en la arquitectura se reflean en la GUI casi de forma instantánea.

En la siguiente imagen, podemos ver la información referente a los clientes:

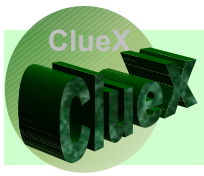


Para un funcionamiento óptimo, se puede **ajustar la velocidad de refresco** en cualquier momento

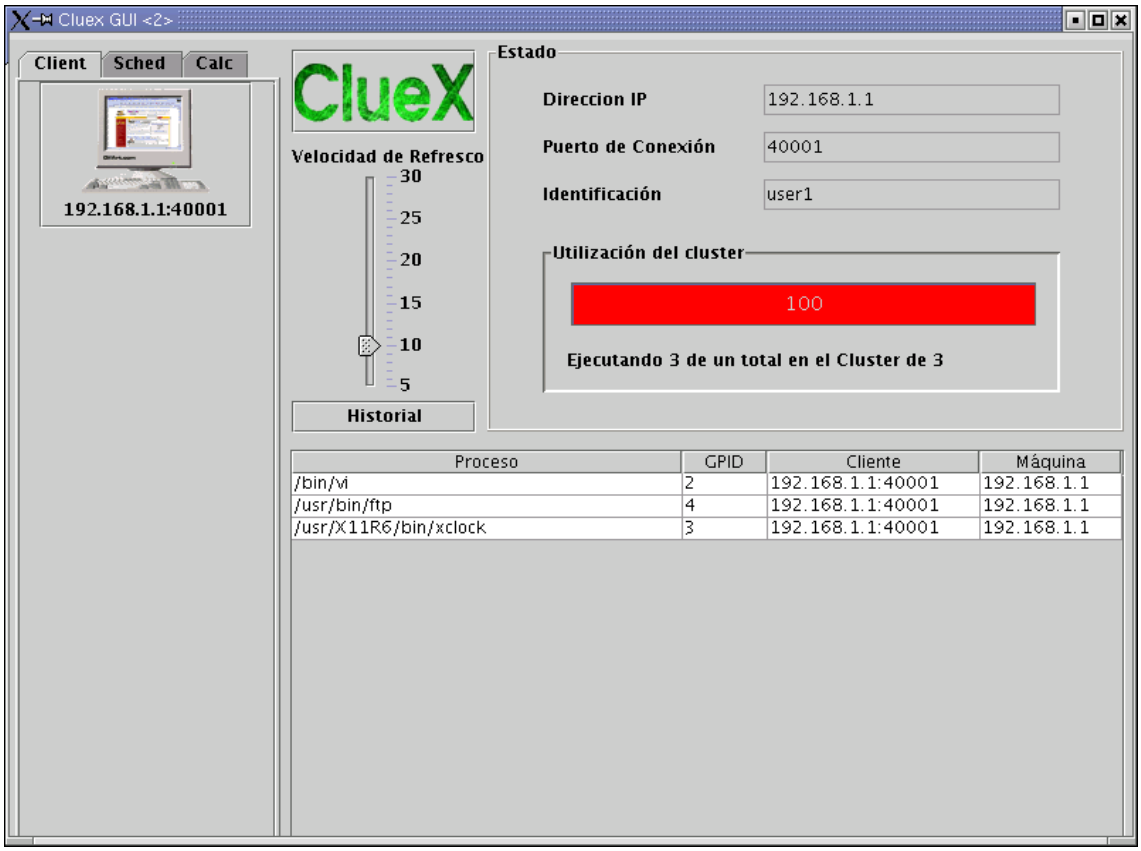
En la mitad inferior, se mostrarían los **procesos que se encontraran en ejecución en ese momento**. También es posible visualizar un **historial global**, pulsando el botón correspondiente.



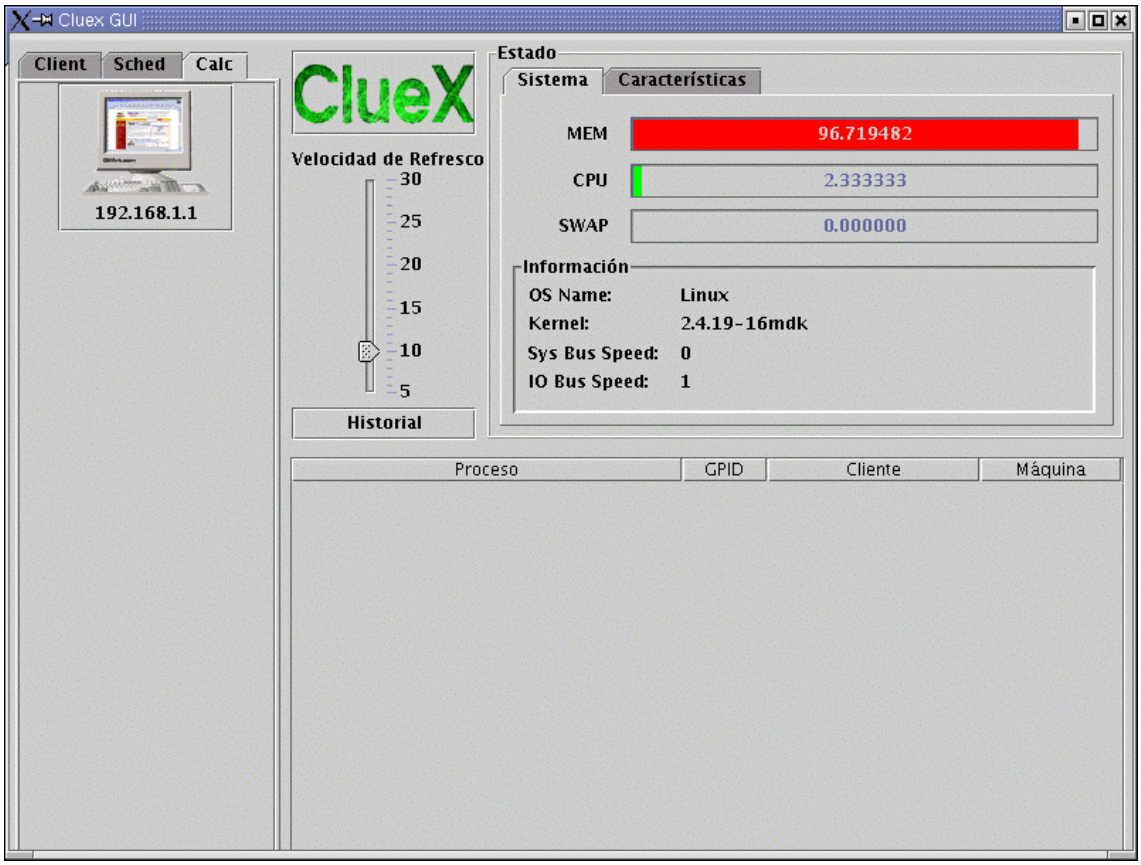
En la imagen superior podemos ver la lista con el **historial,** y, en la de la página siguiente, la **información de los procesos en ejecución**.



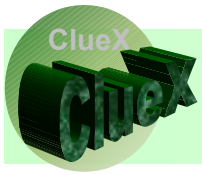
6.- La interfaz gráfica.



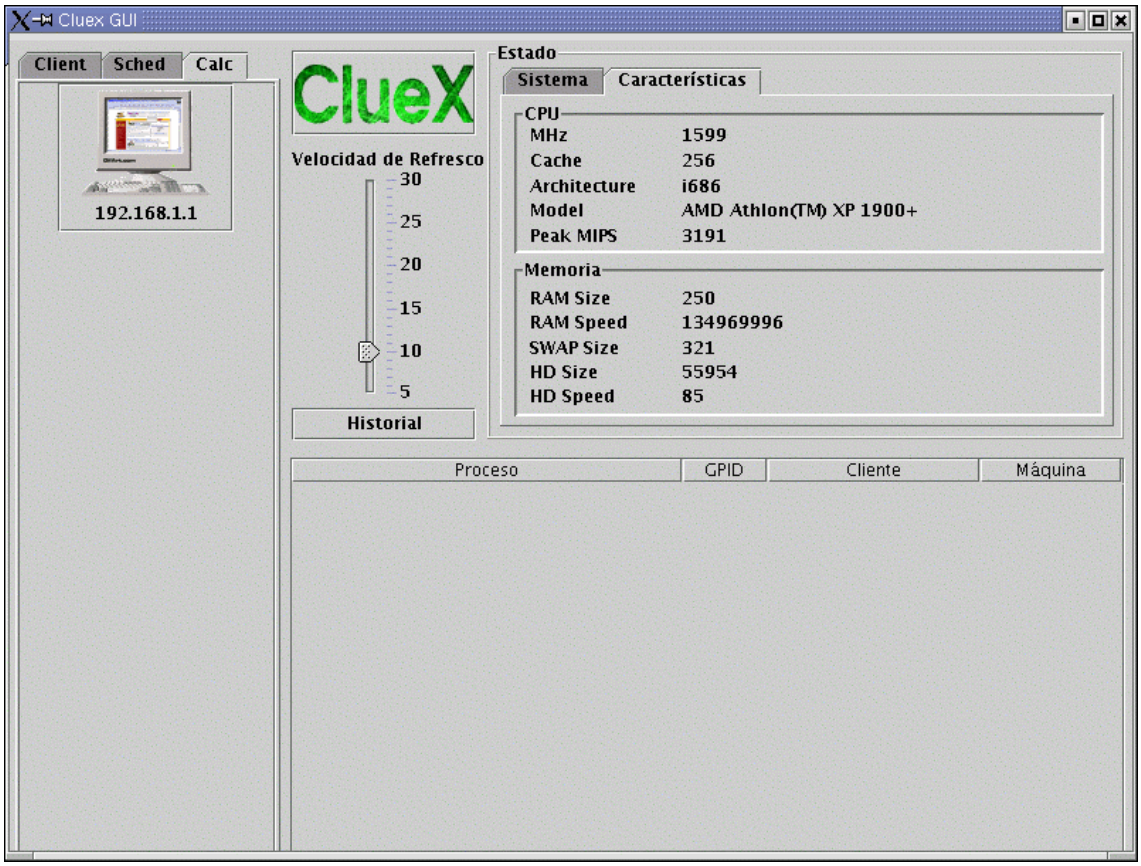
6.3.2.- Información de las estaciones de cálculo.



En todo momento, aparece la **información dinámica y la carga actual** del sistema. Pero si pulsamos sobre la pestaña de la parte superior derecha "características", también estará a nuestra disposición la **información estática**:



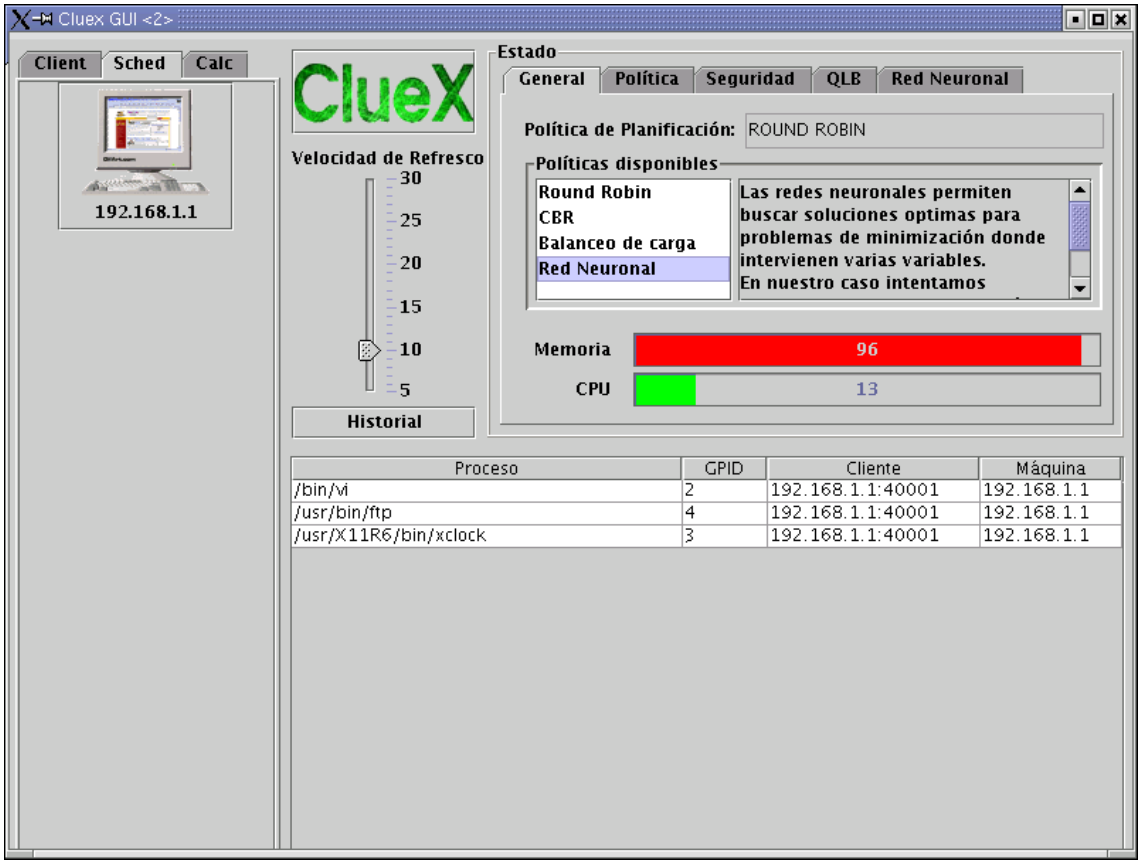
6.- La interfaz gráfica.



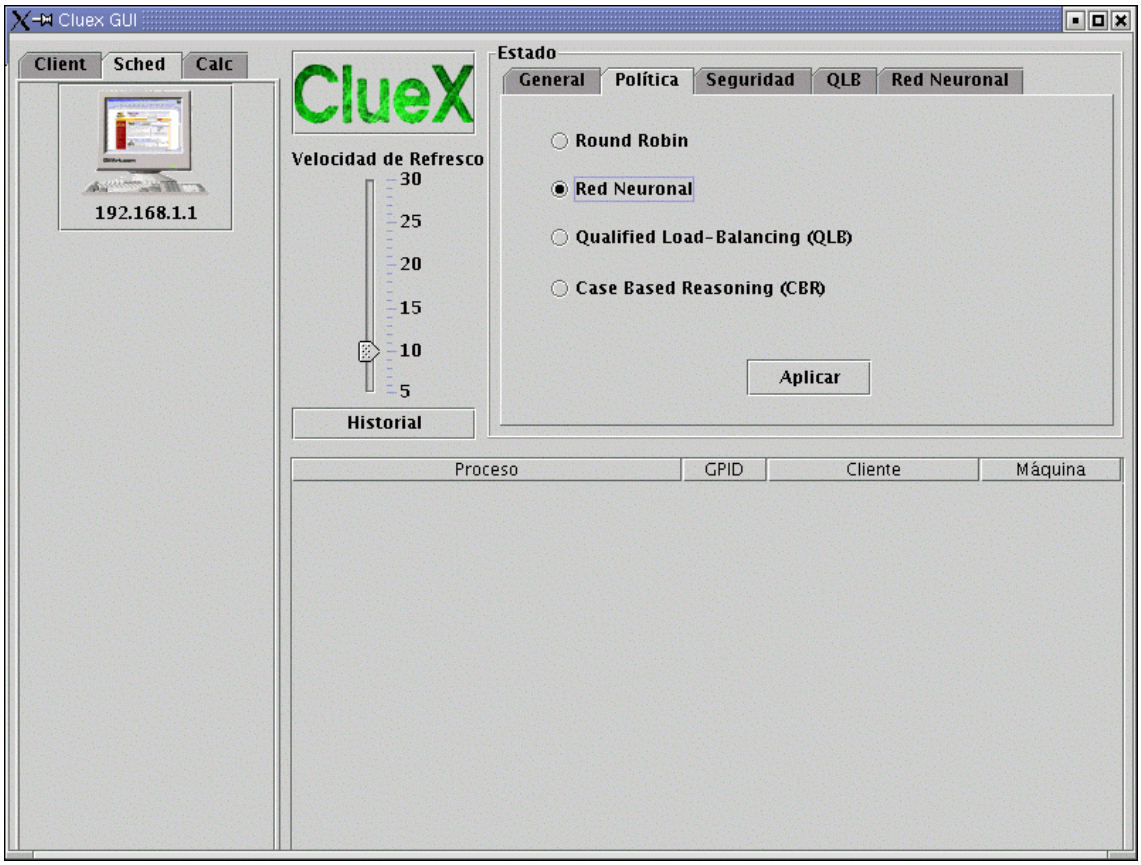
6.3.3.- Información del planificador.

Por supuesto, ésta será la parte de la GUI más importante para el administrador, ya que es la que permite configurar el sistema conforme a sus necesidades.

En la pestaña principal se muestra la **política de planificación actual**, una **descripción de las políticas** y la **carga de CPU y memoria** de la máquina en la que se ejecuta el planficador:

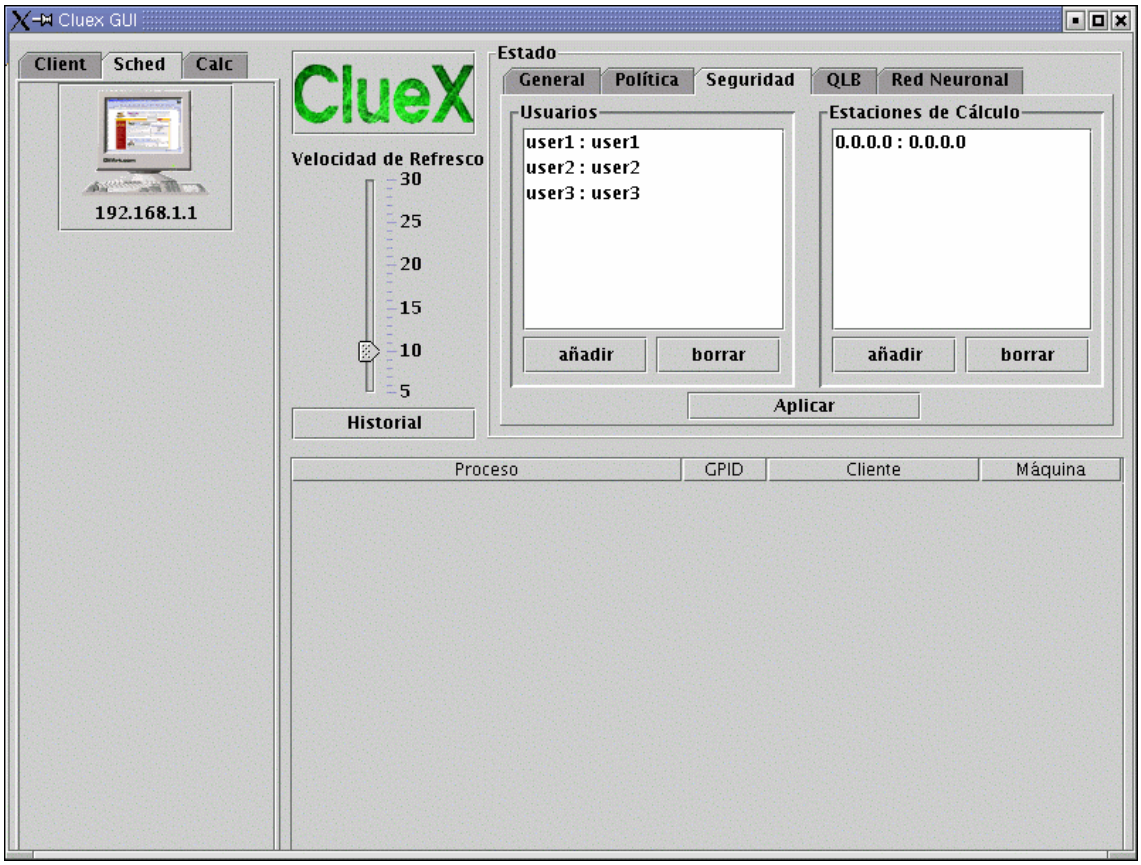


También permite **designar el algoritmo de planificación** de tareas a aplicar:



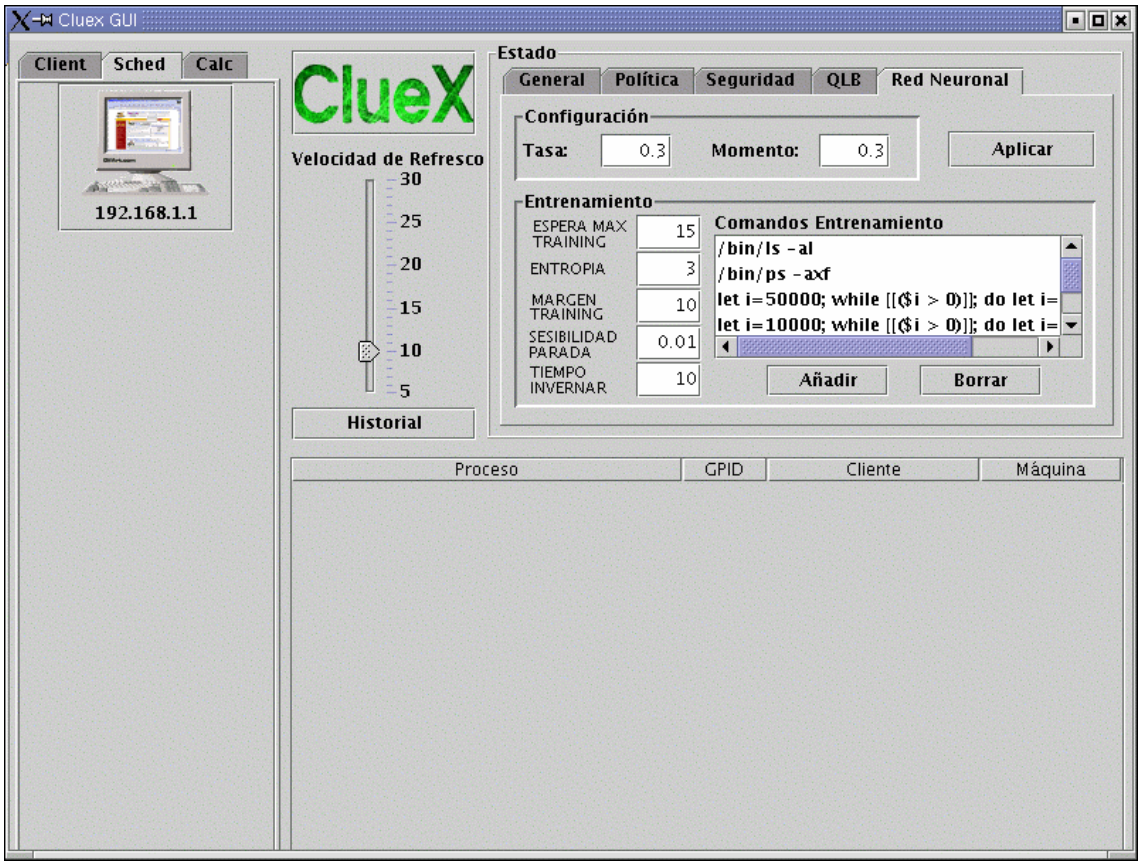
Gracias a la posibilidad que ofrece ClueX de cambiar de política de planificación en cualquier momento, el administrador, lo hará, si así lo desea.

Se puede visualizar y modificar las características de seguridad:

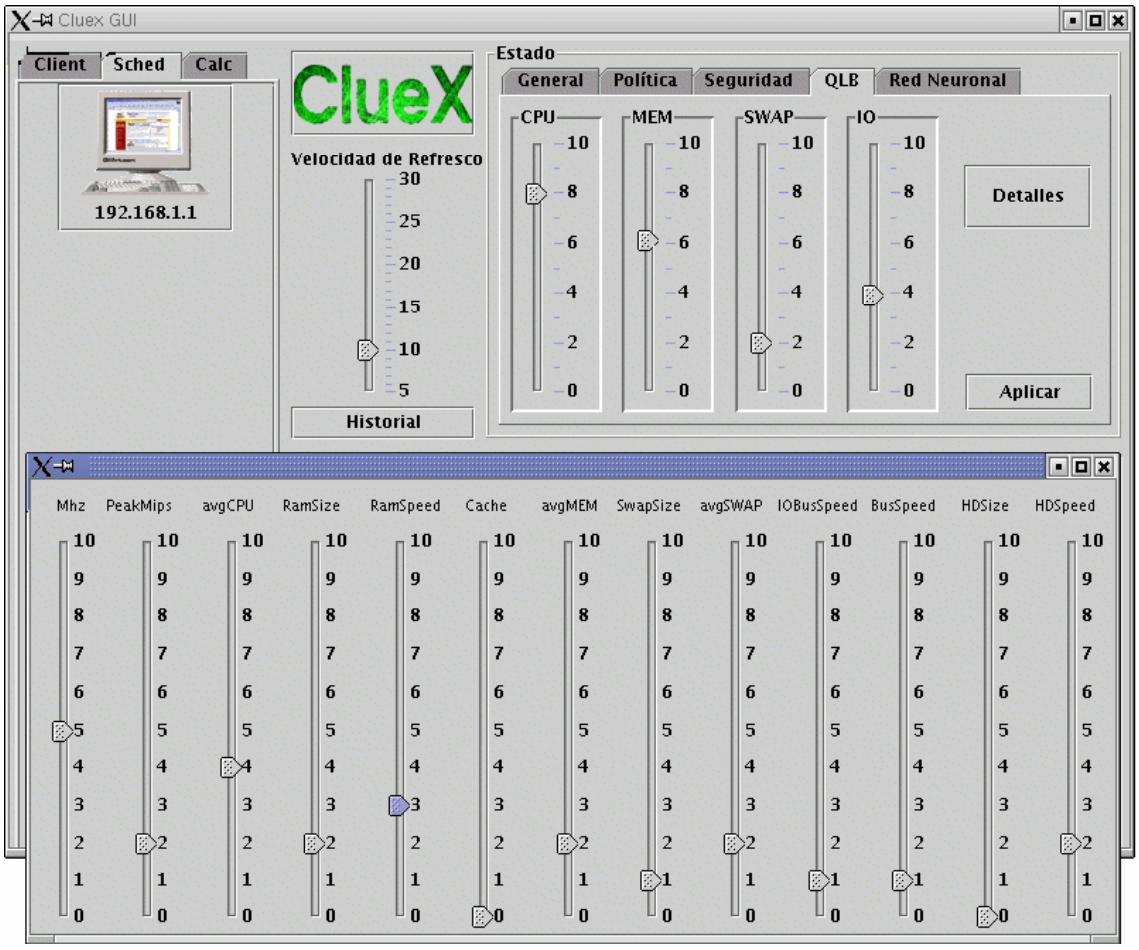


De nuevo, ClueX, de permitir el cambio de las características de los algoritmos de planificación “en caliente”. Por ello, la GUI permite cambiar las características de aquellos algoritmos que admiten configuración específica. Éstas son:

Las **Redes Neuronales**, con los valores de configuración correspondientes:



Y el **Balanceo de Carga**, con posibilidad de modificar el valor de cada peso por cada característica:



6.4.- Comunicación con la GUI desde el planificador

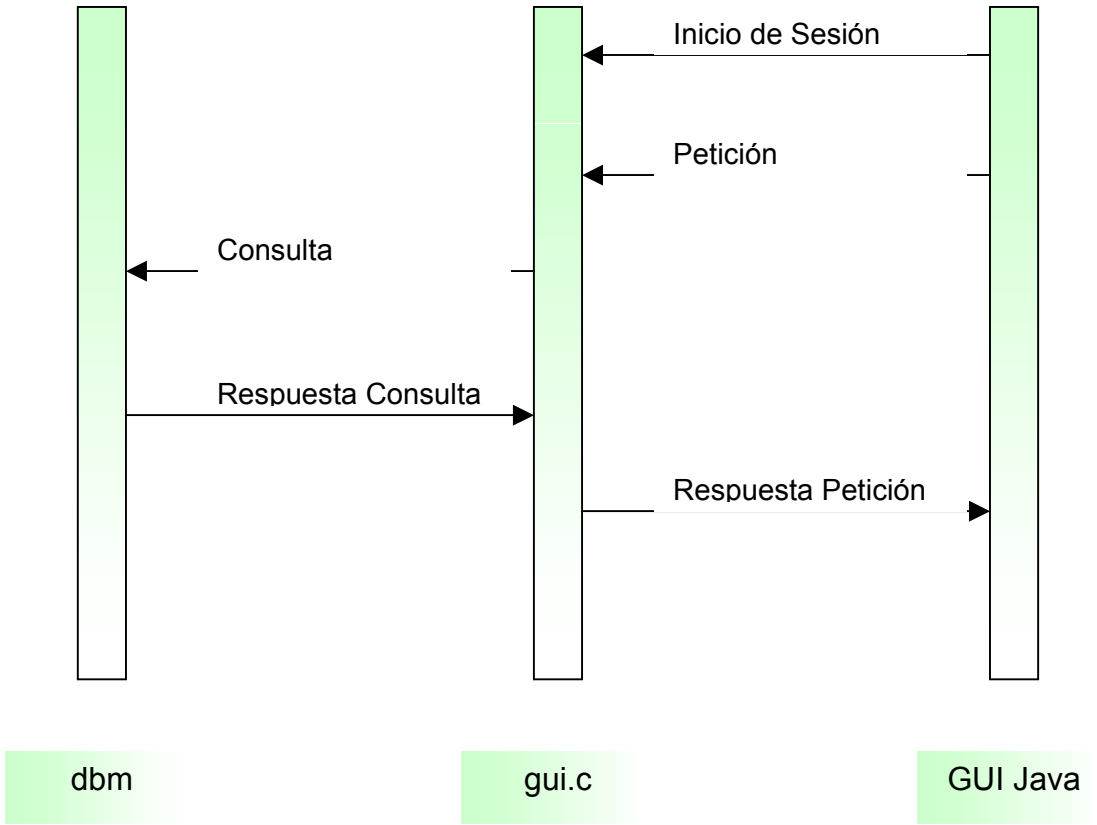
Las comunicación con la GUI Java desde el planificador se realiza a través de un proceso implementado en el archivo *gui.c.* Este proceso se encarga de escuchar en un puerto dedicado las peticiones de sesión y el tratamiento de las peticiones realizadas.

Para la transimisión de los datos utiliza la librería *libCrypt* para su **encriptado y desencriptado**. Aparte, cuando el proceso gui, recibe los datos se ayuda de las funciones auxiliares definidas en el módulo *xml.c* para el procesado de las consultas.

Su integración en el planificador se realiza de una forma natural debido a la **arquitectura basada en colas de mensajes** de éste. Una vez procesada la consulta, se envia a la cola del dbm la petición. En el dbm se implementa un método encargado de realizar dichas consultas a las bases de datos y devolver la información utilizando otra cola de mensajes específica del proceso gui.

De este modo, al recibir la consulta se envía un mensaje al dbm y se espera su respuesta escuchando la otra cola de mensajes. Cada mensaje enviado o recibido, lleva el pid del proceso gui que realiza la consulta, permitiendo de este modo tener varias sesiones abiertas a la vez.

Una vez obtenida la información desde el dbm, el proceso gui genera los XML necesarios con la información y los envía a la GUI Java debidamente encriptados.

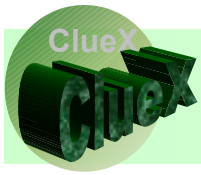


MÓDULO VIII

PRUEBAS

DE

RENDIMIENTO



7.- Pruebas de rendimiento.

7.1.- Introducción.

Principalmente, son 2 las razones que nos han llevado a realizar una **batería de pruebas** sobre ClueX:

- Resulta interesante averiguar de qué forma el trabajo realizado con el presente proyecto logra un aumento del rendimiento de un conjunto de máquinas.

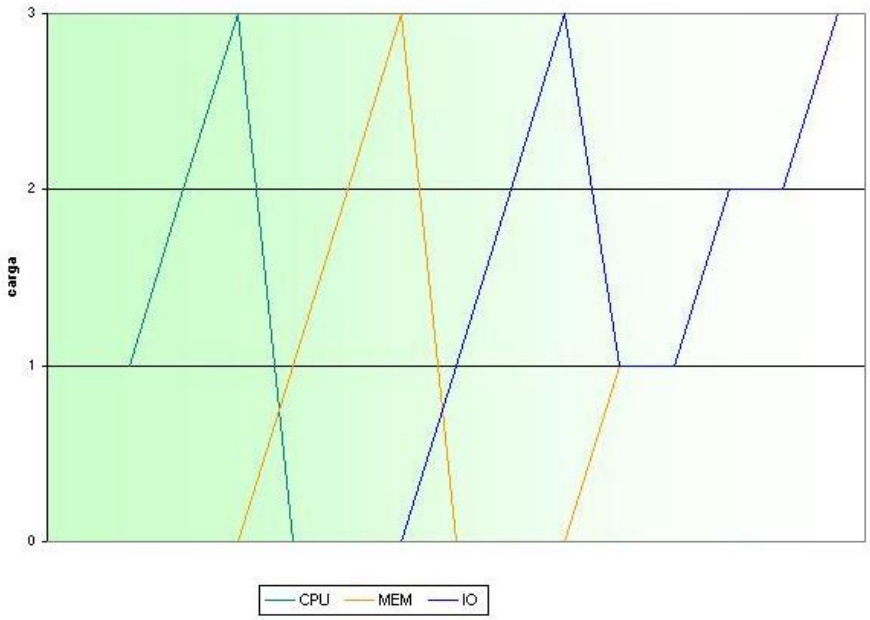
- No menos atractiva es la idea de realizar una comparativa entre los algoritmos de planificación implementados, y saber para qué tipos de procesos son más eficientes unos algoritmos que otros.

7.2.- Metodología

Se ha diseñado un **script** que lanza periódicamente una serie de procesos con alta carga de CPU, memoria y entrada salida, y se han ido **midiendo los tiempos de ejecución** de los mismos para cada tipo de algoritmo de planificación.

El sistema de prueba consistía en 3 estaciones de cálculo de diversas características, conectadas a un planificador, y un cliente desde donde se solicitaba la ejecución de los scripts de benchmarking.

Concretamente, se han definido procesos con **3 niveles de carga: bajo, medio y alto**. A lo largo de un espacio de tiempo relativamante corto, se lanzan procesos de los 3 tipos, de forma que, globalmente, nuestro cluster se vea sometido a una carga como la indicada por el siguiente gráfico:

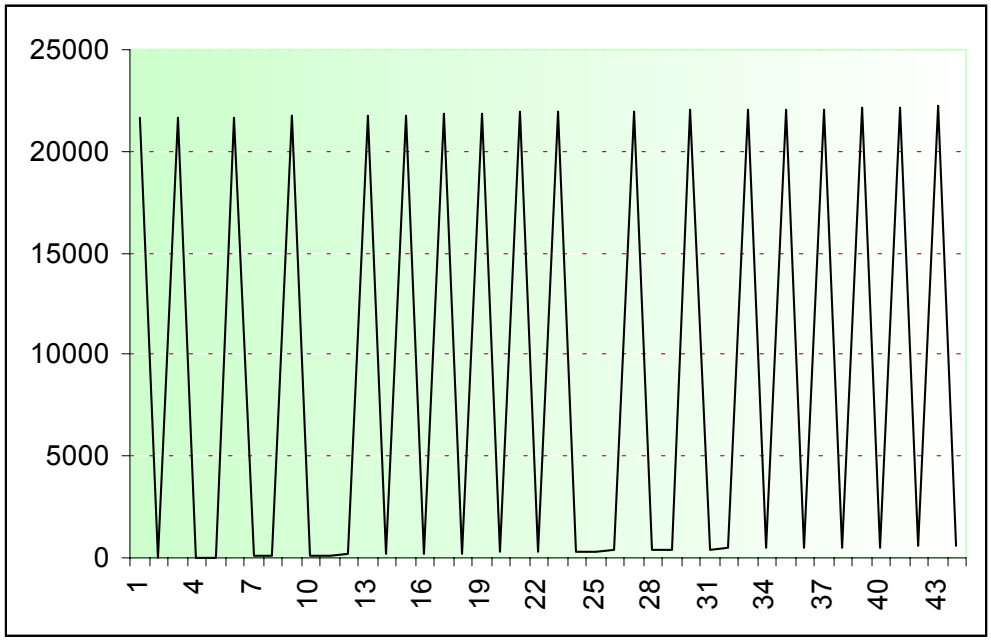


NOTA: En las últimas partes del proceso de pruebas, ejecutamos los 3 tipos al mismo tiempo (COU, Memoria y E/S), y al mismo nivel. No es apreciable en la gráfica, ya que las líneas se superponen.

7.3.- Resultados

7.3.1.- Resultados con Round-Robin.

La gráfica siguiente muestra los tiempos de ejecución de los procesos de prueba utilizando el turno rotatorio: **El eje de abscisas muestra el número de proceso ejecutado. El de ordenadas, el tiempo de ejecución en milisegundos:**

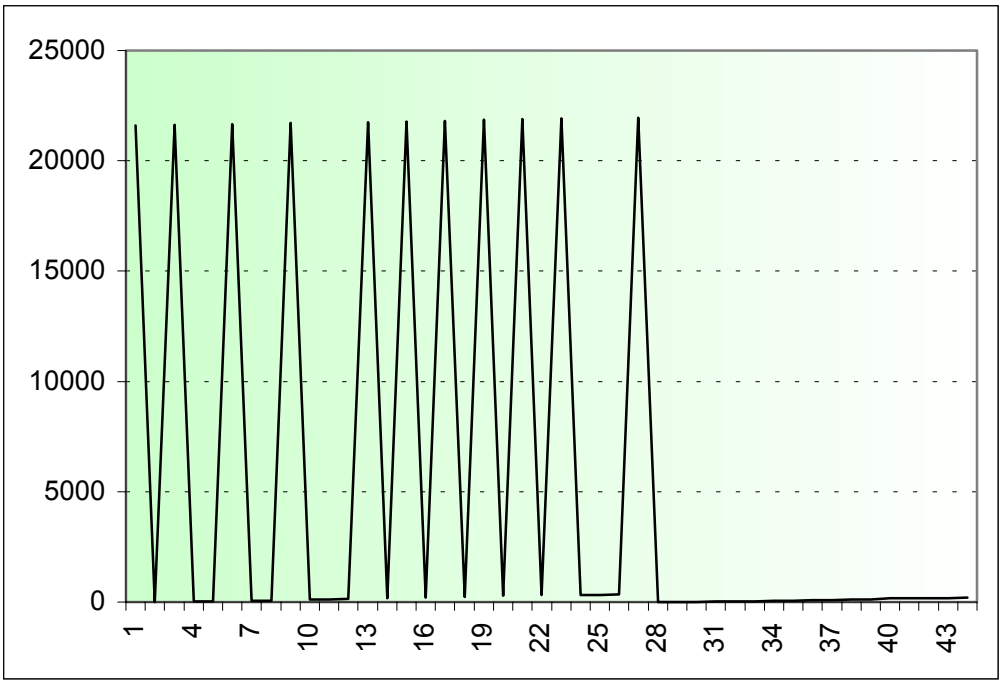


El tiempo total de ejecución de los procesos asciende a 401895 ms

La **forma ondulatoria** de la gráfica tiene su lógica: al tener estaciones de cálculo de muy distintas características, los respectivos tiempos de ejecución de una misma tarea son muy distintos en cada una de ellas. Además, debido a la sobrecarga a la que se somete las tareas, los tiempos de ejecución van siendo mayores en cada máquina.

7.3.2.- Resultados con CBR.

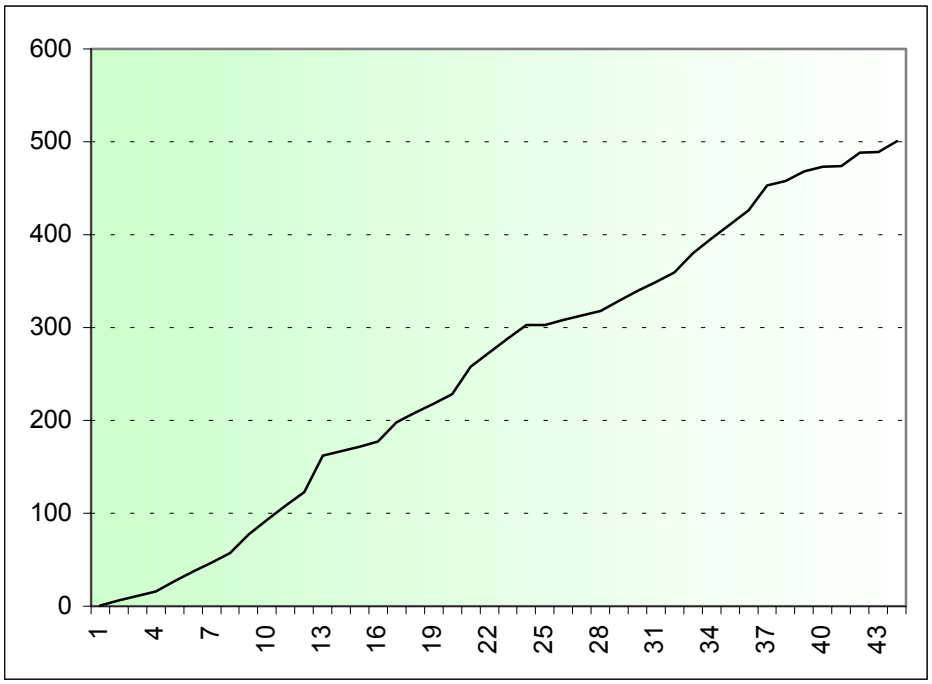
La gráfica resultante muestra claramente las características propias de este tipo de planificación: se distinguen **dos fases**; en la primera el comportamiento es similar al del Round-Robin, ya que no existen casos validados, y el sistema debe seguir “aprendiendo”. En la segunda fase, ya existen casos validados, y, en lugar de aprender, el sistema envía directamente las tareas a las mismas máquinas que fueron enviadas en aquellos casos óptimos.



El tiempo total de procesamiento es de **249208 milisegundos.**

7.3.3.- Resultados con la Red Neuronal.

Despues de un periodo razonable de aprendizaje, la red neuronal ha terminado asignando los procesos a las máquina más potentes. La siguiente gráfica muestra los tiempos de proceso de las tareas.

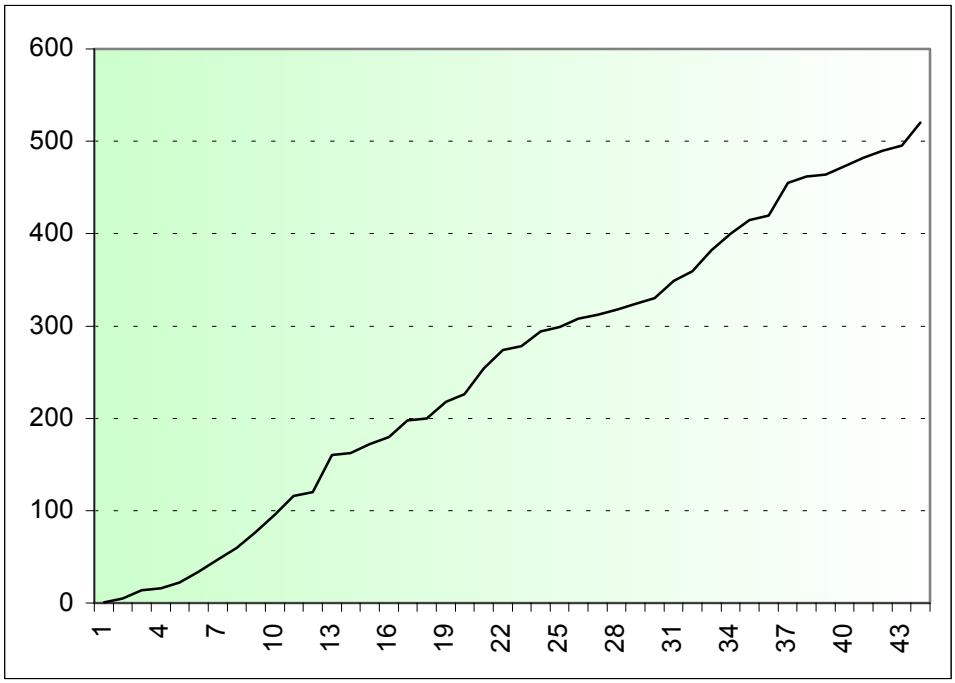


Los procesos son asignados a las mejores máquinas, que van siendo sobrecargadas con la llegada de nuevas tareas. El tiempo total de procesamiento es de 11290 milisegundos.

7.3.4.- Resultados con Balanceo de Carga.

El balanceo de carga se limita a enviar siempre los procesos a la máquina con mejores características de acuerdo con el fichero de configuración. En nuestro sistema, la diferencia de rendimiento de las estaciones de cálculo es muy distinta, y prácticamente todos los procesos se llevan a la misma máquina, a pesar de la sobrecarga que ésta va sufriendo.

La gráfica obtenida es muy parecida a la anterior: la llegada masiva de procesos a una misma estación de cálculo, provoca su sobrecarga y su **gradual pérdida de rendimiento**.



El acumulado de tiempos de ejecución de los procesos asciende en este caso a **11537 milisegundos.**

7.4.- Conclusiones.

Se ha comprobado claramente la **ventaja de los algoritmos basados en heurística** sobre un algoritmo ciego como el Round Robin.

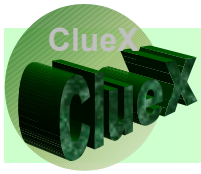
A partir de esta idea, parece lógico admitir que resulta conveniente para el administrador de ClueX **conocer las características de las estaciones de cálculo que tiene a su disposición** a la hora de decidir el algoritmo de planificación que debe activar, así como **las características que demandan los procesos** que va a llevar a cabo.

MÓDULO VIII

LOS MÓDULOS

MÁS

RELEVANTES



8.1.- Dbm.c

```
/** \file dbm.c
*
*   \author Proyecto ClueX: Daniel Navas-Parejo, Juan Antonio Recio, Luis Domínguez
*   \date 05.01.2003
*   \version 1.15
*/
#include <gdbm.h>
#include "datacollect.h"
#include "dbm.h"
#include "gui.h"
#include "AIEngine.h"
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <pthread.h>
//#include <sys/time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
//#include <time.h>

#include "memwatch.h"



#define VERBOSE 1


/* Descriptores de las tablas de la BD:
 * calc_dbf --> BD de estaciones de cálculo
 * client_dbf --> BD de clientes
 * proc_dbf --> BD de procesos ejecutándose actualmente
 * prochist_dbf --> BD de estadísticas de procesos ya ejecutados
 */

GDBM_FILE calc_dbf,client_dbf,proc_dbf,prochist_dbf;
key_t mkey;
int receptionq, schedq, guiq;


/**
*       Insercion de un gpid en una lista de gpids
*       \param gpid = gpid a insertar en la lista
*       \param gpidlist = lista de gpids donde se insereta el nuevo gpid
*/
void insert_gpid(unsigned long int gpid,gpid_list* gpidlist) {


        gpidlist->size++;
        if (gpidlist->size == 1) {
                gpidlist->gpids = (unsigned long int*) malloc(sizeof(unsigned long int));
                }
        else {
                gpidlist->gpids = (unsigned long int*) realloc(gpidlist->gpids,(gpidlist->size)*sizeof(unsigned long int));
                }
        gpidlist->gpids[gpidlist->size-1] = gpid;
        }

int delete_gpid(unsigned long int gpid, gpid_list* gpidlist) {
```

```
	int i=0;
	int seguir = 1;

	if (gpidlist->size <= 1) {
		gpidlist->size = 0;
		}
	else {
		while ((i<gpidlist->size) && seguir) {
			if (gpidlist->gpids[i] == gpid) {
				seguir = 0;
				}
			}
		memmove(gpidlist->gpids+i,gpidlist->gpids+i+1,gpidlist->size-i-1);
		gpidlist->size--;
		}
	return (seguir != 0);
	}

/**
*	lista de gpids: Inicializacion
*
*	\return lista de gpids inicializada
*/
gpid_list new_gpid_list(void) {
	gpid_list glist;

	glist.size = 0;
	return glist;
	}


int create_data(int dtype,void *data,datum *gdata){

	machineinfo mach;
	sysinfo sys;
	meminfo mem;
	cpuinfo cpu;
	int* dptrint;
	char* dptrchar;
	int longitud,i;
	float *dptrfloat;
	unsigned long int *dptrulong;
	new_process_msg *newp;
	new_client_msg *newcl;
	history_info *hinfo;


	switch(dtype) {
		case KEY_CTABLE:
			gdata->dsize = sizeof(unsigned long int);
			gdata->dptr = (char*)malloc(sizeof(unsigned long int));
			dptrulong = (unsigned long int *) gdata->dptr;
			dptrulong[0] = *((unsigned long int *)data);
			return 0;
		case DATA_CTABLE:
			mach = *((machineinfo*)data);
			sys = mach.sys;
			mem = sys.mem;
			cpu = sys.cpu;
			gdata->dptr =
(char*)malloc(sizeof(sys)+strlen(sys.osname)+strlen(sys.kernelver)+strlen(sys.cpu.arch)+strlen(sys.cpu.model)+ 64 + 100);
			strcpy(gdata->dptr,"");
			dptrint = (int *)gdata->dptr;
			longitud = 0;
			dptrint[0] = cpu.mhz;
			dptrint[1] = cpu.csize;
			longitud = longitud + sizeof(int)*2;
			dptrchar = gdata->dptr + longitud;
			dptrchar[0]=0;
			sprintf(dptrchar,"%s%c%s%c",cpu.arch,SEPARATOR,cpu.model,SEPARATOR);
			longitud += 2 + strlen(cpu.arch) + strlen(cpu.model);
			dptrint = (int *)(dptrchar + strlen(cpu.arch) + strlen(cpu.model) + 2);
			dptrint[0] = cpu.peakmips;
			dptrint[1] = mem.ramsize;
			dptrint[2] = mem.ramspeed;
			dptrint[3] = mem.swapsize;
```

```
		dptrint[4] = mem.hdsize;
		dptrint[5] = mem.hdspeed;
		longitud += 6*sizeof(int);
		dptrchar = gdata->dptr + longitud;
		dptrchar[0] = 0;
		sprintf(dptrchar,"%s%c%s%c",sys.osname,SEPARATOR,sys.kernelver,SEPARATOR);
		longitud += 2 + strlen(sys.osname) + strlen(sys.kernelver);
		dptrint = (int *)(dptrchar + strlen(sys.osname) + strlen(sys.kernelver) + 2);
		dptrint[0] = sys.sysbusspeed;
		dptrint[1] = sys.iobusspeed;
		longitud += 2*sizeof(int);
		dptrfloat = (float*)(dptrint + 2);
		dptrfloat[0] = mach.din.cpuload;
		dptrfloat[1] = mach.din.mem.average_mem;
		dptrfloat[2] = mach.din.mem.average_swap;
		longitud += 3*sizeof(float);
		dptrint  = (int*)(dptrfloat + 3);

		dptrchar =(char*) dptrint;
		memcpy(dptrchar, mach.clave, 64);
		dptrchar += 64;
		dptrint = (int*) dptrchar;
		longitud+=64;

		dptrint[0] = mach.gpids.size;
		for (i=0;i<mach.gpids.size;i++) {
			dptrint[i+1] = mach.gpids.gpids[i];
			}
		longitud += (i+1)*sizeof(int);
		gdata->dsize = longitud+1;
		return 0;
case KEY_XTABLE:
		gdata->dsize = sizeof(unsigned long int);
		gdata->dptr = (char*)malloc(sizeof(unsigned long int));
		memcpy(gdata->dptr, data, sizeof(unsigned long int));
		return 0;
case DATA_XTABLE:
		newp = (new_process_msg *) data;
		gdata->dptr = (char*) malloc(sizeof(new_process_msg));
		gdata->dsize= sizeof(new_process_msg);
		memcpy(gdata->dptr, data, sizeof(new_process_msg));
		return 0;
case KEY_CLTABLE:
		gdata->dsize = sizeof(unsigned long int) + sizeof(int);
		gdata->dptr  = malloc(gdata->dsize);
		newcl = (new_client_msg * ) data;
		dptrulong = (unsigned long int *) (gdata->dptr);
		*dptrulong= newcl->ipaddr;
		dptrulong++;
		dptrint = (int*) dptrulong;
		*dptrint = newcl->port;
		return 0;
case DATA_CLTABLE:
		gdata->dsize = sizeof(new_client_msg);
		gdata->dptr  = malloc(gdata->dsize);
		memcpy(gdata->dptr, data, gdata->dsize);
		return 0;
case KEY_PTABLE:
		gdata->dsize = 34;
		gdata->dptr = (char*)malloc(34);
		memcpy(gdata->dptr, data, 34);
		return 0;
case DATA_PTABLE:
		hinfo = (history_info *) data;

		gdata->dptr = (char*)malloc( sizeof(history_info) );
		gdata->dsize= sizeof (history_info);
		memcpy(gdata->dptr, hinfo, sizeof(history_info));
		/*
		gdata->dptr = (char*)malloc( (4*sizeof(unsigned long int)) + 256 +30);
		gdata->dsize = (4*sizeof(unsigned long int)) + 256 + 30;
		dptrulong = (unsigned long int *) (gdata->dptr);
		dptrulong[0] = hinfo->ipaddr_client;
		dptrulong[1] = hinfo->ipaddr_calc;
		dptrulong[2] = hinfo->tiempo;
		dptrulong[3] = hinfo->gpid;
```

```
                        dptrchar =((char *)gdata->dptr)+ (4*sizeof(unsigned long int));
                        memcpy(dptrchar,hinfo->orden,256);
                        dptrchar += 256;
                        memcpy(dptrchar,hinfo->fecha, 30);
                        */
                        return 0;
                default:
                        return -1;
                }
        }

int extract_data(int dtype,void *data,datum gdata) {

        machineinfo* mach;
        sysinfo* sys;
        int longitud,*dptrint,i;
        char *dptrchar;
        float *dptrfloat;
        unsigned long int *dptrulong, *dptrulong2;
        new_process_msg *newp;
        history_info *hinfo;

        switch(dtype) {
                case KEY_CTABLE:
                        dptrulong = (unsigned long int *) (gdata.dptr);
                        dptrulong2 = (unsigned long int *) data;
                        dptrulong2[0] = *dptrulong;
                        return 0;
                case DATA_CTABLE:
                        mach = (machineinfo *) data;
                        sys = &(mach->sys);
                        dptrint = (int*)(gdata.dptr);
                        longitud = 0;
                        sys->cpu.mhz = dptrint[0];
                        sys->cpu.csize = dptrint[1];
                        longitud += 2*sizeof(int);
                        dptrchar = gdata.dptr + longitud;
                        //sys->cpu.arch = (char *) malloc(strlen(dptrchar)+1);
                        strcpy(sys->cpu.arch,dptrchar);
                        longitud += strlen(dptrchar)+1;
                        dptrchar = gdata.dptr + longitud;
                        //sys->cpu.model = (char *) malloc(strlen(dptrchar)+1);
                        strcpy(sys->cpu.model,dptrchar);
                        longitud += strlen(dptrchar)+1;
                        dptrint = (int*)(dptrchar + strlen(dptrchar) + 1);
                        sys->cpu.peakmips = dptrint[0];
                        sys->mem.ramsize = dptrint[1];
                        sys->mem.ramspeed = dptrint[2];
                        sys->mem.swapsize = dptrint[3];
                        sys->mem.hdsize = dptrint[4];
                        sys->mem.hdspeed = dptrint[5];
                        longitud += 6*sizeof(int);
                        dptrchar = gdata.dptr + longitud;
                        //sys->osname = (char *) malloc(strlen(dptrchar) + 1);
                        strcpy(sys->osname,dptrchar);
                        longitud += strlen(dptrchar) + 1;
                        dptrchar = gdata.dptr + longitud;
                        //sys->kernelver = (char *) malloc(strlen(dptrchar) + 1);
                        strcpy(sys->kernelver,dptrchar);
                        longitud += strlen(dptrchar) + 1;
                        dptrint = (int*)(dptrchar + strlen(dptrchar) + 1);
                        sys->sysbusspeed = dptrint[0];
                        sys->iobusspeed = dptrint[1];
                        longitud += 2*sizeof(int);
                        dptrfloat = (float*)(dptrint + 2);
                        mach->din.cpuload = dptrfloat[0];
                        mach->din.mem.average_mem = dptrfloat[1];
                        mach->din.mem.average_swap = dptrfloat[2];
                        longitud += 3*sizeof(float);
                        dptrint = (int*)(dptrfloat + 3);

                        dptrchar = (char*)dptrint;
                        memcpy(mach->clave, dptrchar, 64);
                        dptrchar += 64;
                        dptrint = (int*) dptrchar;
                        longitud += 64;
```

```
                        /*
                        mach->gpids.size = dptrint[0];
                        for (i=0;i<mach->gpids.size;i++) {
                                mach->gpids.gpids[i] = dptrint[i+1];
                                }
                        */
                        mach->gpids.size = 0;
                        for (i=0;i<dptrint[0];i++) {
                                insert_gpid(dptrint[i+1], &(mach->gpids));
                                }

                        longitud += (i+1)*sizeof(int);
                        //data = sys;
                        return 0;
                case KEY_XTABLE:
                        dptrulong = (unsigned long int *) data;
                        *dptrulong = *((unsigned long int *) (gdata.dptr));
                        return 0;
                case DATA_XTABLE:
                        newp = (new_process_msg *) data;
                        memcpy( newp, gdata.dptr, sizeof(new_process_msg));
                        return 0;
                case DATA_CLTABLE:
                        memcpy(data, gdata.dptr, gdata.dsize);
                        return 0;
                case KEY_PTABLE:
                        memcpy(data, gdata.dptr, gdata.dsize);
                        return 0;
                case DATA_PTABLE:
                        hinfo = (history_info *) data;
                        memcpy( hinfo, gdata.dptr, sizeof(history_info));
                        /*
                        dptrulong = (unsigned long int *) (gdata.dptr);
                        hinfo->ipaddr_client = dptrulong[0];
                        hinfo->ipaddr_calc = dptrulong[1];
                        hinfo->tiempo = dptrulong[2];
                        hinfo->gpid   = dptrulong[3];
                        dptrchar =((char *)gdata.dptr)+(4*sizeof(unsigned long int));
                        memcpy(hinfo->orden, dptrchar,256);
                        dptrchar += 256;
                        memcpy(hinfo->fecha, dptrchar, 30);
                        */
                         return 0;
                default:
                        return -1;
                }
        }

int new_calc_station(unsigned long int ipaddr,sysinfo sys, char* clave) {
        datum key,gdata;
        machineinfo mach;
      sched_msg msg;

        //sys.cpu.model = (char *)malloc(MAXCAD);
        //sys.cpu.arch = (char *)malloc(MAXCAD);
        //sys.osname = (char *)malloc(MAXCAD);
        //sys.kernelver = (char *)malloc(MAXCAD);
        //strcpy(sys.cpu.model,"");
        //strcpy(sys.cpu.arch,"");
        //strcpy(sys.osname,"");
        //strcpy(sys.kernelver,"");

        create_data(KEY_CTABLE,(void *) &ipaddr, &key);

        mach.sys = sys;
        mach.din.cpuload = 0;
        mach.din.mem.average_mem = 0;
        mach.din.mem.average_swap = 0;
        mach.gpids = new_gpid_list();
        bzero(mach.clave,64);
        strcpy(mach.clave, clave);

        create_data(DATA_CTABLE,(void *) &mach, &gdata);
        gdbm_store(calc_dbf,key,gdata,GDBM_REPLACE);

        msg.mtype = CALC_STATIONS_CHANGE;
```

```
        msgsnd(schedq,&msg,sizeof(sched_msg),0);

        return 0;
        }

int delete_calc_station(unsigned long int ipaddr) {
        datum key;
        struct in_addr inet;
    sched_msg msg;

        create_data(KEY_CTABLE,(void *) &ipaddr, &key);
        gdbm_delete(calc_dbf,key);
    inet.s_addr = ipaddr;
        printf("<-//-> Borrada estacion de calculo %s\n", inet_ntoa(inet));

        msg.mtype = CALC_STATIONS_CHANGE;
        msgsnd(schedq,&msg,sizeof(sched_msg),0);

    return 0;
    }


int new_dinamic(unsigned long int ipaddr,dinamicload dinamic) {
        datum key,gdata;
        machineinfo mach;
    sched_msg msg;

        create_data(KEY_CTABLE,(void *) &ipaddr, &key);
        gdata = gdbm_fetch(calc_dbf,key);
        extract_data(DATA_CTABLE,(machineinfo *) &mach, gdata);
        mach.din = dinamic;
        create_data(DATA_CTABLE,(void *) &mach, &gdata);
        gdbm_store(calc_dbf,key,gdata,GDBM_REPLACE);

        msg.mtype = NEW_DINAMIC_INFO;
        msgsnd(schedq,&msg,sizeof(sched_msg),0);

        return 0;
        }

/**
*       adicion de un nuevo proceso en ejecucion en la base de datos
*
*       \param ipaddrclient = IP del cliente solicitante de la ejecucion de la orden
*       \param ipaddrcal = IP de la estacion de calculo que ejecuta la orden
*       \param gpid = pid global de la ejecucion
*/
int new_running_proccess(new_process_msg msg){
        datum key,gdata;
        machineinfo mach;
        //new_process_msg aux;

        if(VERBOSE) printf("DBM. new_runing_process START gpid %ld\n", msg.gpid);
        create_data(KEY_XTABLE, (void *) &msg.gpid, &key);
        create_data(DATA_XTABLE, (void *) &msg, &gdata);
        gdbm_store(proc_dbf,key,gdata,GDBM_REPLACE);
        free(gdata.dptr);
        //gdata = gdbm_fetch (proc_dbf, key);
        //extract_data(DATA_XTABLE, &aux, gdata);
        //printf("Extraido: gpid: %ld\n\n", aux.gpid);
        //free(gdata.dptr);

        create_data(KEY_CTABLE,(void *) &msg.ipaddr_calc, &key);
    gdata = gdbm_fetch(calc_dbf,key);
        extract_data(DATA_CTABLE,(machineinfo *) &mach, gdata);
        insert_gpid(msg.gpid,&mach.gpids);
        create_data(DATA_CTABLE,(void *) &mach,&gdata);
        gdbm_store(calc_dbf,key,gdata,GDBM_REPLACE);

        if(VERBOSE) printf("DBM. new_runing_process STOP \n");
        return 0;
        }

unsigned long int end_running_process(unsigned long int tiempo, unsigned long int gpid, unsigned long int *decision, char* fecha) {
        datum key,gdata;
        machineinfo mach;
```

```
        new_process_msg newpm;
      unsigned long int gpid_caso;
        history_info hinfo;
        char * dptrchar;


        //Primero se borra de la tabla de procesos en ejecucion
        create_data(KEY_XTABLE, (void *) &gpid, &key);
        gdata = gdbm_fetch(proc_dbf,key);
        extract_data(DATA_XTABLE,(new_process_msg *) &newpm, gdata);
        gpid_caso = newpm.gpid_caso;
        *decision = newpm.estado;
        if (VERBOSE) printf("DBM end_runn_proc DECISION es %ld\n", *decision);
        gdbm_delete(proc_dbf,key);
        free(key.dptr);
        free(gdata.dptr);
        gdata.dsize = 0;
        key.dsize = 0;


        //Luego se actualiza la tabla de estaciones de calculo
        //indicando que esa estacion ya no ejecuta ese proceso
        create_data(KEY_CTABLE,(void *) &newpm.ipaddr_calc, &key);
        gdata = gdbm_fetch(calc_dbf,key);
        if (VERBOSE) printf("DBM end_runn_proc; GDATA dsize = %d\n",gdata.dsize);
        extract_data(DATA_CTABLE,(machineinfo *) &mach, gdata);
        delete_gpid(gpid,&(mach.gpids));
        create_data(DATA_CTABLE,(void *) &mach,&gdata);
        gdbm_store(calc_dbf,key,gdata,GDBM_REPLACE);
        free(key.dptr);
        free(gdata.dptr);
        gdata.dsize = 0;
        key.dsize = 0;


        //Para acabar guardamos la info en el historial
        hinfo.ipaddr_client = newpm.ipaddr_client;
        hinfo.puerto_client = newpm.puerto_client;
        hinfo.ipaddr_calc   = newpm.ipaddr_calc;
        hinfo.tiempo        = tiempo;
        hinfo.gpid          = gpid;
        memcpy(hinfo.orden, newpm.orden, 256);
        bzero(hinfo.fecha, 30);
        strcpy(hinfo.fecha, fecha);
        //printf("Proceso: tiempo: %ld, gpid: %ld, orden: %s, fecha: %s\n", hinfo.tiempo, hinfo.gpid, hinfo.orden, hinfo.fecha);
        dptrchar = malloc( 34 );
        memcpy(dptrchar  , &gpid,        4);
        memcpy(dptrchar+4, hinfo.fecha, 30);
        create_data(KEY_PTABLE, (void*) dptrchar,  &key);
        create_data(DATA_PTABLE,(void*) &hinfo, &gdata);
        gdbm_store(prochist_dbf, key, gdata, GDBM_REPLACE);
        free(gdata.dptr);
        free(key.dptr);
        free(dptrchar);
        gdata.dsize=0;
        key.dsize = 0;
        return gpid_caso;
        }

int new_client(new_client_msg msgClient)
{
        datum key, data;
        create_data(KEY_CLTABLE, &msgClient, &key);
        create_data(DATA_CLTABLE, &msgClient, &data);
        gdbm_store(client_dbf, key, data, GDBM_REPLACE);

        return 0;
}

int delete_client(delete_client_msg msgDeleteClient)
{
        datum key;
        struct in_addr dir;

        create_data(KEY_CLTABLE, &msgDeleteClient, &key);
        gdbm_delete(client_dbf, key);
```

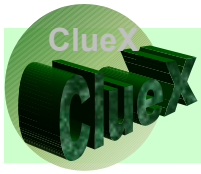
```
	dir.s_addr = msgDeleteClient.ipaddr;
	printf("<-//-> Borrando cliente %s:%d   de ClueX\n",inet_ntoa(dir), msgDeleteClient.port);
	return 0;
}

void read_all_table(int table) {
	datum gdata,key,nextkey;
	machineinfo mach;
	unsigned long int ipaddr;
	char str_ipaddr[20];
	struct in_addr inet;
	info_estacion_calculo *macharray;
	int i,j;
	sched_msg msg;

	key = gdbm_firstkey(calc_dbf);

	macharray = NULL;
	i = 0;
      while (key.dptr) {
		gdata = gdbm_fetch(calc_dbf,key);
		extract_data(KEY_CTABLE,&ipaddr,key);
		inet.s_addr = ipaddr;
		strcpy(str_ipaddr,inet_ntoa(inet));
			extract_data(DATA_CTABLE,&mach,gdata);
			macharray = (info_estacion_calculo *) realloc(macharray,(i+1)*sizeof(info_estacion_calculo));
			macharray[i].mach = mach;
			macharray[i].ip_addr = ipaddr;
			/*printf("\nDatos correspondientes a la estación de cálculo %s\n",str_ipaddr);
			printf("CPU MHZ:\t\t\t%d\n",mach.sys.cpu.mhz);
			printf("CPU CACHE:\t\t\t%d\n",mach.sys.cpu.csize);
			printf("PEAK MIPS:\t\t\t%d\n",mach.sys.cpu.peakmips);
			printf("RAM SIZE:\t\t\t%d\n",mach.sys.mem.ramsize);
			printf("SWAP SIZE:\t\t\t%d\n",mach.sys.mem.swapsize);
			printf("HD SIZE:\t\t\t%d\n",mach.sys.mem.hdsize);
			printf("CPU MODEL:\t\t\t%s\n",mach.sys.cpu.model);
			printf("CPU ARCHITECTURE:\t\t%s\n",mach.sys.cpu.arch);
			printf("RUNNING OS:\t\t\t%s\n",mach.sys.osname);
			printf("KERNEL VERSION:\t\t\t%s\n",mach.sys.kernelver);
			printf("CPU USAGE:\t\t\t%.2f\n",mach.din.cpuload);
			printf("RAM USAGE:\t\t\t%.2f\n",mach.din.mem.average_mem);
			printf("SWAP USAGE:\t\t\t%.2f\n",mach.din.mem.average_swap);*/
			nextkey = gdbm_nextkey(calc_dbf, key);
	free(key.dptr);
	key = nextkey;
			free(gdata.dptr);
			i++;
	}
	for (j=0; j<i; j++) {
			msg.mtype = READ_RESULT_MSG;
			msg.data.contenido = macharray[j];
			msg.data.contenido.tamanio_restante = i - j - 1;
			msgsnd(schedq,&msg,sizeof(sched_msg),0);
			}
	}


void guiQuery( gui_query gq, unsigned long int ipConsulta )
{
	datum key, nextkey, gdata;
	unsigned long int ipaddr;
	gui_answer gans;
	char hostname[256];
	struct hostent * host;
	machineinfo mach;
	unsigned long int * puli;
	int port;
	int* pi;
	char aux[256];
	FILE *file;
	long int  total, usado;
	new_process_msg newpm;
	history_info hinfo;
```

```
gans.gui_id = gq.client_id;
gans.query_type = gq.query_type;

switch (gq.query_type)
{
	case GUI_QUERY_GLOBAL:
		//printf("dbm. consulta global iniciada. Enviando estaciones de calculo\n");
		//primero las estaciones de calculo
		key = gdbm_firstkey(calc_dbf);
		while (key.dptr)
		{
			extract_data(KEY_CTABLE,&ipaddr, key);
			gans.answer_type = GUI_ANS_GLOBAL_CALCSTATION_IP;
			gans.isLast = 0;
			gans.data.ip = ipaddr;
			nextkey = gdbm_nextkey(calc_dbf, key);
			//free(key.dptr);
			key = nextkey;
			msgsnd (guiq, &gans, sizeof(gui_answer),0);
		}
		//luego los clientes
		key = gdbm_firstkey(client_dbf);
		while (key.dptr)
		{
			puli = (unsigned long int*)key.dptr;
			ipaddr = *puli;
			puli++; pi = (int*)puli;
			port = *pi;
			gans.answer_type = GUI_ANS_GLOBAL_CLIENT_IP;
			gans.isLast = 0;
			gans.data.client_ip.ip =   ipaddr;
			gans.data.client_ip.port = port;
			nextkey = gdbm_nextkey(client_dbf, key);
			//free(key.dptr);
			key = nextkey;
			msgsnd (guiq, &gans, sizeof(gui_answer),0);
		}
		//el planificador
		gethostname(hostname, 256);
		host = gethostbyname(hostname);
		gans.data.ip = *(long*)host->h_addr ;
		gans.answer_type = GUI_ANS_GLOBAL_SCHEDULER_IP;
		key = gdbm_firstkey(proc_dbf); //para saber si es el ultimo msg
		gans.isLast = !key.dptr;
		msgsnd(guiq, &gans, sizeof(gui_answer),0);
		// por ultimo la lista de procesos en ejecucion
		while (key.dptr)
		{
			gdata = gdbm_fetch(proc_dbf,key);
			extract_data(DATA_XTABLE,(new_process_msg *) &newpm, gdata);
			gans.answer_type = GUI_ANS_GLOBAL_EXECPROCESS;
			gans.data.execProc = newpm;
			//free(gdata.dptr);
			nextkey = gdbm_nextkey(proc_dbf, key);
			//free(key.dptr);
			key = nextkey;
			if (!key.dptr)
				gans.isLast = 1;
			else gans.isLast = 0;
			msgsnd (guiq, &gans, sizeof(gui_answer),0);
		}
		printf("dbm. consulta global acabada\n");
		break;

	case GUI_QUERY_CALCSTATION:

		create_data(KEY_CTABLE,(void *) &ipConsulta, &key);
		gdata = gdbm_fetch(calc_dbf,key);
		extract_data(DATA_CTABLE,(machineinfo *) &mach, gdata);
		gans.data.machInfo = mach;
		gans.answer_type = GUI_ANS_CALCSTATION;
		gans.isLast = 1;
		msgsnd(guiq, &gans, sizeof(gui_answer),0);
		break;
```

```
                case GUI_QUERY_CLIENT:
                        break;

                case GUI_QUERY_SCHEDULER:
                        printf("DBM GUI. Scheduler start\n");
                        //Hallamos la info del planificador:
                        gans.data.infSched.policy=sched_policy;
                        //Hallamos el porcentaje de CPU libre
                        sprintf(aux,"vmstat > ");
                        strcat(aux,VMFILE);
                        system(aux);
                        if (NULL == (file = fopen(VMFILE,"r")))
                                perror("Error al abrir el fichero\n");
                        //saltamos las 2 primeras lineas:
                        fgets(aux,256,file);
                        fgets(aux,256,file);
                        fgets(aux,256,file);
                        strcpy(aux,aux+76);
                        fclose(file);
                        unlink(VMFILE);
                        gans.data.infSched.stat_cpu = 100 - strtol(aux,NULL,10);
                        //Hallamos la memoria libre de la que dispone
                        sprintf(aux,"free -o  > ");
                        strcat(aux,FREEFILE);
                        system(aux);
                        if (NULL == (file = fopen(FREEFILE,"r")))
                                perror("Error al abrir el fichero\n");
                        //saltamos la primera linea
                        fgets(aux,256,file);
                        fgets(aux,256,file);

                        strcpy(aux, aux+5);
                        total = strtol(aux,NULL,10);
                        strcpy(aux, aux+25);
                        usado = strtol(aux,NULL,10);
                        printf("DBM GUI. total mem: %ld, usado: %ld", total, usado);
                        gans.data.infSched.stat_mem = 100 - ( usado* 100 /total );
                        fclose(file);
                        unlink(FREEFILE);
                        gans.answer_type=GUI_ANS_SCHEDULER;
                        gans.isLast=1;
                        msgsnd(guiq, &gans, sizeof(gui_answer),0);
                        printf("DBM GUI. Scheduler finish\n");
                        break;
                case GUI_QUERY_HISTORY:
                        key = gdbm_firstkey(prochist_dbf);
                        while (key.dptr)
                        {
                                gdata = gdbm_fetch(prochist_dbf,key);
                                extract_data(DATA_PTABLE, &hinfo, gdata);
                                printf("DBM GUI. Proceso extraido: tiempo: %ld, gpid: %ld, orden: %s, fecha: %s\n",
hinfo.tiempo, hinfo.gpid, hinfo.orden, hinfo.fecha);
                                gans.answer_type = GUI_ANS_HISTORY;
                                gans.data.histoInfo = hinfo;
                                free(gdata.dptr);
                                nextkey = gdbm_nextkey(prochist_dbf, key);
                                free(key.dptr);
                                key.dptr = nextkey.dptr;
                                key.dsize = nextkey.dsize;
                                if (key.dptr == 0)
                                        gans.isLast = 1;
                                else gans.isLast = 0;
                                msgsnd (guiq, &gans, sizeof(gui_answer),0);
                        }
                        break;

        }
}

void dbm(void *arg) {

        dbm_msg mensaje;
      sched_msg mensaje_s;

        puts("");
```

```
	errno = 0;
	calc_dbf = gdbm_open(PATH_CTABLE,512,GDBM_NEWDB,0x0777,NULL);
	perror("GDBM Open Calc DBF");
	errno = 0;
	client_dbf = gdbm_open(PATH_CLTABLE,512,GDBM_NEWDB,0x0777,NULL);
	perror("GDBM Open Client DBF");
	errno = 0;
	proc_dbf = gdbm_open(PATH_XTABLE,512,GDBM_NEWDB,0x0777,NULL);
	perror("GDBM Open Process DBF");
	errno = 0;
	prochist_dbf = gdbm_open(PATH_PTABLE,512,GDBM_WRCREAT,0x0777,NULL);
	perror("GDBM Open Process Historic DBF");
	mkey = ftok(".",'b');
	if (-1 == (receptionq=msgget(mkey,0777))) {
		perror("Error en la creación de la cola de mensajes del DBM");
	}
	mkey = ftok(".",'a');
	if (-1 == (schedq=msgget(mkey,0777))) {
		perror("Error en la creación de la cola de mensajes del DBM");
	}
	mkey = ftok(".",'g');
	if (-1 == (guiq=msgget(mkey,0777))){
		perror("Error en la creación de la cola de mensajes de la GUI");
	}

	while(1) {
		msgrcv(receptionq,&mensaje,1024,0,0);
		switch(mensaje.mtype) {
			case STATIC_MSG:
				new_calc_station(mensaje.data.staticm.ipaddr,mensaje.data.staticm.sys,
mensaje.data.staticm.clave);
				break;
			case DINAMIC_MSG:
				new_dinamic(mensaje.data.dinamicm.ipaddr,mensaje.data.dinamicm.din);
				break;
			case NEW_CLIENT_MSG:
				new_client(mensaje.data.newcm);
				break;
			case NEW_RUNNING_PROCESS_MSG:
				new_running_proccess(mensaje.data.newpm);
				break;
			case READ_ALL_CALC_STATION_MSG:
				read_all_table(DATA_CTABLE);
				break;
			case DELETE_CALC_MSG:
				delete_calc_station(mensaje.ipaddr);
				break;
			case TIME_ANSWER_MSG:
				if(VERBOSE) printf("DBM termina proceso\n");

				mensaje_s.mtype = END_PROCESS_EXECUTION;
				//mensaje_s.data.time.gpid_caso = mensaje.data.timem.gpid;
				mensaje_s.data.time.gpid_caso = end_running_process(mensaje.data.timem.time,
mensaje.data.timem.gpid, &mensaje_s.data.time.decision, mensaje.data.timem.fecha);
				mensaje_s.data.time.tiempo = mensaje.data.timem.time;
				strcpy(mensaje_s.data.time.orden,mensaje.data.timem.orden);
				msgsnd(schedq,&mensaje_s,sizeof(sched_msg),0);
				if(VERBOSE) printf("DBM mensaje fin proceso enviado\n");

				break;
			case DELETE_CLIENT_MSG:
				delete_client(mensaje.data.delcm);
				break;
			case GUI_QUERY:
				guiQuery(mensaje.data.guiqm, mensaje.ipaddr);
			}
		}
	}

void terminar(int arg) {

	gdbm_close(calc_dbf);
	gdbm_close(client_dbf);
	gdbm_close(proc_dbf);
	gdbm_close(prochist_dbf);
	msgctl(receptionq,IPC_RMID,NULL);
```

```
	msgctl(guiq,IPC_RMID,NULL);
	printf("\nClueX Data Base Manager Finished\n");
	exit(0);
	}

void terminar_DBM_fault(int arg) {
	fprintf(stderr,"\nDBM SEGMENTATION FAULT\n");
	gdbm_close(calc_dbf);
	gdbm_close(client_dbf);
	gdbm_close(proc_dbf);
	gdbm_close(prochist_dbf);
	msgctl(receptionq,IPC_RMID,NULL);
	msgctl(guiq,IPC_RMID,NULL);
	fflush(stderr);
	exit(-1);
	}
void nada_dbm(int arg) {
	}

int mainDBM() {
	struct hostent * host;
	char h[256];
	gethostname(h, 256);
	host = gethostbyname(h);

	signal(SIGTERM,terminar);
	signal(SIGINT,terminar);
	signal(SIGSEGV,terminar_DBM_fault);
	signal(SIGUSR1,nada_dbm);
	mwInit();
	dbm((void *)NULL);
	exit(0);
	}
```



8.2.- AIEngine.c

```
/** \file AIEngine.c
*
*   \author Proyecto ClueX: Daniel Navas-Parejo, Juan Antonio Recio, Luis Domínguez
*   \date 05.01.2003
*   \version 1.15
*/
#include "datacollect.h"
#include "dbm.h"
#include "puertos.h"
#include "transferencia.h"
#include "red.h"
#include "AIEngine.h"
#include "cppexport.h"
#include "qlb.h"
#include "nnet.h"
#include "nnettraining.h"
#include "cluexCBR.h"

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#define VERBOSE 0



int dbmq, schedq;
info_estacion_calculo *maquinas;
cola_peticion current_proc;
int sched_policy;
unsigned long int ip_anterior = 0;
unsigned long int gpid = 0;
int trainq, nnetq, pid_training, pid_nnet;

int segmentoAI;
void * memAI;
int NNET_TRAINING;


info_estacion_calculo* recibir_machineinfo(info_estacion_calculo primero) {
        sched_msg mensaje_s;
    long int i=1;
    info_estacion_calculo *macharray;
        int tamanio_restante;
    char* cmd;
    dbm_msg mensaje_dbm;


        tamanio_restante = primero.tamanio_restante;
        macharray = (info_estacion_calculo *) malloc((tamanio_restante + 1)*sizeof(info_estacion_calculo));
        macharray[0] = primero;

        if ((sched_policy == NEURONAL_NET) && NNET_TRAINING)
```

```
	{
		mensaje_s.mtype = READ_RESULT_MSG;
		mensaje_s.data.contenido = macharray[0];
		msgsnd(trainq, &mensaje_s, sizeof(mensaje_s),0);
	}

	while (tamanio_restante > 0){
		msgrcv(schedq,&mensaje_s,sizeof(sched_msg),0,0);
		switch (mensaje_s.mtype) {
			case READ_RESULT_MSG:
				macharray[i] = mensaje_s.data.contenido;
				tamanio_restante = macharray[i].tamanio_restante;
				i++;
				if ((sched_policy == NEURONAL_NET) && NNET_TRAINING)
					msgsnd(trainq, &mensaje_s, sizeof(mensaje_s),0);
				break;
			case NEW_PROCESS_SCHED_MSG:
				current_proc.ultimo++;
				cmd = (char*) malloc(strlen(mensaje_s.data.nuevo_proc.comando)+1);
				cmd[strlen(mensaje_s.data.nuevo_proc.comando)] = 0;
				strcpy(current_proc.peticiones[current_proc.ultimo].comando,cmd);

	strcpy(current_proc.peticiones[current_proc.ultimo].comando,mensaje_s.data.nuevo_proc.comando);
				current_proc.peticiones[current_proc.ultimo].ip_addr = mensaje_s.data.nuevo_proc.ip_addr;
				current_proc.peticiones[current_proc.ultimo].puerto = mensaje_s.data.nuevo_proc.puerto;
				current_proc.peticiones[current_proc.ultimo].idSesion = mensaje_s.data.nuevo_proc.idSesion;
				mensaje_dbm.mtype = READ_ALL_CALC_STATION_MSG;
				msgsnd(dbmq,&mensaje_dbm,sizeof(dbm_msg),0);
				break;
			}
		}

	return macharray;
	}

unsigned long int round_robin(long int tamanio) {
	int i, seguir;

	if (ip_anterior == 0)
	{
		ip_anterior = maquinas[0].ip_addr;
		return maquinas[0].ip_addr;
	}
	else {
		i = 0;
		seguir = 1;
		while((i<tamanio) && (seguir))
		{
			if (maquinas[i].ip_addr == ip_anterior)
				seguir = 0;
			i++;
		}
		ip_anterior = maquinas[i%tamanio].ip_addr;
		return ip_anterior;
		}
	}

/**
*	Realiza la decision del algoritmo de planificacion que se va a utilizar
*	Simplemente, lee la constante y llama a la funcion correspondiente
*	\param tamanio = numero de estaciones de calculo del sistema
*	\return IP de la estacion de calculo seleccionada
*/
unsigned long int decidir(long int tamanio, int indice) {
	unsigned long int ip_addr;
	//unsigned long int gpid_antiguo, estado;
	dbm_msg mensaje;
	struct in_addr inet;
	cabeceraQLB headerQLB;
	double* entrada;
	int i, posicion, tam;
	nnet_msg mensajeRed;

	mensaje.mtype = NEW_RUNNING_PROCESS_MSG;
	mensaje.data.newpm.gpid = gpid;
	mensaje.data.newpm.gpid_caso = gpid;
```

```
	mensaje.data.newpm.estado = 0;
	mensaje.data.newpm.ipaddr_client = current_proc.peticiones[indice].ip_addr;
	//mensaje.data.newpm.ipaddr_calc = ip_addr;
	mensaje.data.newpm.puerto_client = current_proc.peticiones[indice].puerto_asignado;
	bzero(mensaje.data.newpm.orden, 256);
	strcpy(mensaje.data.newpm.orden, current_proc.peticiones[indice].comando);


	switch (sched_policy) {
		case ROUND_ROBIN:
			ip_addr = round_robin(tamanio);
			mensaje.data.newpm.ipaddr_calc = ip_addr;
			msgsnd(dbmq,&mensaje,1024,0);

			inet.s_addr = ip_addr;
			if (VERBOSE) printf("Decidido por RR %s\n",inet_ntoa(inet));

			return ip_addr;
			break;
		case CBR:
/*			ip_addr = ejecutaCBR(current_proc.peticiones[indice].comando,gpid,maquinas, &gpid_antiguo);
			inet.s_addr = ip_addr;
			if (ip_addr == 0)
				ip_addr = round_robin(tamanio);
				inet.s_addr = ip_addr;
				if (VERBOSE) printf("Decidido por RR %s\n",inet_ntoa(inet));
			else
				if (VERBOSE) printf("Decidido por CBR %s\n",inet_ntoa(inet));
			estado = estadoMaquina(current_proc.peticiones[indice].comando,gpid_antiguo,ip_addr,maquinas, tamanio);
			if (VERBOSE) printf("FIN Estado de maquina %ld\n",estado);
			mensaje.mtype = NEW_RUNNING_PROCESS_MSG;
			mensaje.data.newpm.gpid = gpid;
			mensaje.data.newpm.gpid_caso = gpid_antiguo;
			mensaje.data.newpm.estado = estado;
			mensaje.data.newpm.ipaddr_client = current_proc.peticiones[indice].ip_addr;
			mensaje.data.newpm.ipaddr_calc = ip_addr;
			mensaje.data.newpm.puerto_client = current_proc.peticiones[indice].puerto_asignado;
			bzero(mensaje.data.newpm.orden, 256);
			strcpy(mensaje.data.newpm.orden, current_proc.peticiones[indice].comando);
			if (VERBOSE) printf("Enviando mensaje NEW_PROCESS a DBM desde AI\n");
			msgsnd(dbmq,&mensaje,1024,0);
			return ip_addr;
*/			break;
		case QLB:
			configuraCabeceraQLB(&headerQLB,maquinas,tamanio);
			ip_addr = qlb((int)tamanio,maquinas,headerQLB);
			inet.s_addr = ip_addr;
			mensaje.data.newpm.ipaddr_calc = ip_addr;
			msgsnd(dbmq,&mensaje,1024,0);
			if (VERBOSE) printf("Decidido por QLB %s\n",inet_ntoa(inet));
			return ip_addr;
			break;
		case NEURONAL_NET:
			entrada = memAI;
			for (i=0; i<tamanio; i++)
			{
				entrada[i*3]   = maquinas[i].mach.din.cpuload / 100;
				entrada[i*3+1] = maquinas[i].mach.din.mem.average_mem / 100;
				entrada[i*3+2] = maquinas[i].mach.din.mem.average_swap / 100;
			}
			tam = tamanio;
			//printf("AIEngine. Ejecutando red Neuronal START. Tam entrada: %d\n", tam);
			mensajeRed.mtype = NNET_MTYPE_PETICIONES;
			mensajeRed.peticion = NNET_EJECUTA;
			mensajeRed.numMaquinas = tam;
			msgsnd(nnetq, &mensajeRed, sizeof(mensajeRed), 0);
			msgrcv(nnetq, &mensajeRed, sizeof(mensajeRed), NNET_MTYPE_RESPUESTA_EJECUCION, 0);
			posicion = mensajeRed.posicion;
			//printf("AIEngine. Ejecutando red Neuronal STOP\n");


			if(posicion==-1)
				ip_addr = round_robin(tamanio);
			else
				ip_addr = maquinas[posicion].ip_addr;
			inet.s_addr = ip_addr;
```

```
                                printf("Decidido por RedNeuronal %s, Posicion: %d\n",inet_ntoa(inet), posicion);
                                mensaje.data.newpm.ipaddr_calc = ip_addr;
                                msgsnd(dbmq,&mensaje,1024,0);

                                return ip_addr;
                                break;
                        default:
                                return round_robin(tamanio);
                                break;
                        }
        return 0;
        }


/**
*       Establecimiento de la comunicacion estacion de calculo-cliente, para que
*       se realice la ejecucion de la orden.
*       AIEngine se comunica con la estacion de calculo para darle la informacion del cliente.
*       \param ipdestino = ip de la estacion de calculo que realizara la ejecucion
*       \param k = indice de la peticion en curso
*/
void realiza_tunel(unsigned long int ipdestino,int k, unsigned long int gpid_proc) {
      info_nuevo_cliente info;
      struct in_addr inet;
      inet.s_addr =  ipdestino;

          if(VERBOSE) printf("Realizatunel: ipdestino= %d cp.puerto = %d cp.ip_addr =
%d\n",(int)ipdestino,(int)current_proc.peticiones[k].puerto,(int)current_proc.peticiones[k].ip_addr);
      info.puerto = current_proc.peticiones[k].puerto;
      info.ipaddr = current_proc.peticiones[k].ip_addr;
          info.gpid = gpid_proc;
          info.modoEntrenamiento = 0;
          bzero(info.comandoEntrenamiento, 128);
          if(VERBOSE)printf("info.ipaddr = %d\n", (int)info.ipaddr);
      enviarUDP(inet_ntoa(inet),PUERTO_SERVER, &info, sizeof(info_nuevo_cliente));
      }



void iniciarRedNeuronal()
{
        key_t mkey;

        NNET_TRAINING = 1;
        mkey = ftok(".",'n');
        if (-1 == (nnetq=msgget(mkey,IPC_CREAT | 0777)))
                perror("Error en la creación de la cola de mensajes de la red neuronal");
        mkey = ftok("..",'j');
        if ((segmentoAI = shmget(mkey, MAX_MAQUINAS*3*sizeof(double),IPC_CREAT | 0777)) == -1)
                perror("Error en el acceso al segmento de memoria compartida AIEngine");
        memAI = shmat(segmentoAI, 0, 0);
        mkey = ftok(".",'t');
        if (-1 == (trainq=msgget(mkey,0777 | IPC_CREAT)))
                perror("Error en la creación de la cola de mensajes de entrenamiento");
        pid_nnet = fork();
        if (pid_nnet == 0)
                redNeuronal();
        pid_training = fork();
        if (pid_training == 0)
                entrenamientoRedNeuronal();
}

void terminarRedNeuronal()
{
        msgctl(nnetq, IPC_RMID, 0);
        shmctl(segmentoAI, IPC_RMID, 0);
        msgctl(trainq,IPC_RMID,NULL);
        kill(pid_nnet,SIGTERM);
        kill(pid_training, SIGTERM);

}

/**
*       Implementacion del motor de IA
*
*       \param arg = algoritmo de planificacion a aplicar
```

```
*/
void AIEngine(void *arg) {

	sched_msg mensaje_s;
	dbm_msg mensaje_dbm;
	long int tamanio;
	int algoritmo, k;
	unsigned long int ip_destino;
	key_t mkey;
	info_estacion_calculo primero;
	int ajustar_red = 0;
	nnet_msg mensajeRed;

	sched_policy = *((int *)arg);
	switch(sched_policy) {
	case ROUND_ROBIN:
		fprintf(stderr,"Politica de planificacion ROUND ROBIN\n");
		break;
	  case CBR:
		fprintf(stderr,"Politica de planificacion CASE-BASED REASONING\n");
			break;
		case QLB:
			fprintf(stderr,"Politica de planificacion QUALIFIED LOAD-BALANCING\n");
		break;
	case NEURONAL_NET:
		fprintf(stderr,"Politica de planificacion NEURONAL NET\n");
		break;
		}
/* Válido
	if (sched_policy == CBR)
		inicializaArbol();
*/
	if (sched_policy == NEURONAL_NET)
		iniciarRedNeuronal();


	current_proc.ultimo = -1;
	algoritmo = *((int *)arg);
	mkey = ftok(".",'b');
	if (-1 == (dbmq=msgget(mkey,0777))) {
		perror("Error en el acceso a la cola de mensajes del DBM");
		exit(-1);
		}
	mkey = ftok(".",'a');
	if (-1 == (schedq=msgget(mkey,0777))) {
		perror("Error en el acceso a la cola de mensajes del Planificador");
		exit(-1);
		}

	maquinas = 0;

	while(1) {
		msgrcv(schedq,&mensaje_s,sizeof(sched_msg),0,0);
		if(VERBOSE) printf("Mensaje recibido\n");
		switch(mensaje_s.mtype) {
			case READ_RESULT_MSG:
				primero = mensaje_s.data.contenido;
				tamanio = primero.tamanio_restante + 1;
				if(maquinas != 0){ free(maquinas); maquinas =0;}
				maquinas = recibir_machineinfo(primero);
				if ((sched_policy == NEURONAL_NET)&& ajustar_red )
				{
					mensajeRed.mtype = NNET_MTYPE_PETICIONES;
					mensajeRed.peticion = NNET_AJUSTA;
					mensajeRed.numMaquinas = tamanio;
					msgsnd(nnetq, &mensajeRed, sizeof(mensajeRed),0);
					kill(pid_training, SIGUSR2); //Despertamos el enrenamiento
					ajustar_red = 0;
				}
				//printf("AI Maquinas recibidas: %ld\n",tamanio);
				for (k=0; k<=current_proc.ultimo; k++) {
					gpid++;
					if(VERBOSE) printf("ANTES decision\n");
					ip_destino = decidir(tamanio,k);
					if(VERBOSE) printf("DESPUES decision\n");
					realiza_tunel(ip_destino,k,gpid);
```

```
                                        if(VERBOSE) printf("Tunel Realizado\n");
                                        }
                                current_proc.ultimo = -1;
                                break;
                        case NEW_PROCESS_SCHED_MSG:
                                if(VERBOSE) printf("AI Nuevo proceso %s\n",mensaje_s.data.nuevo_proc.comando);
                                current_proc.ultimo = 0;
                                current_proc.peticiones[current_proc.ultimo] = mensaje_s.data.nuevo_proc;
                                mensaje_dbm.mtype = READ_ALL_CALC_STATION_MSG;
                                if(VERBOSE) printf("AI Nuevo proceso %s enviando\n",mensaje_s.data.nuevo_proc.comando);
                                msgsnd(dbmq,&mensaje_dbm,1024,0);
                                if(VERBOSE) printf("AI Nuevo proceso %s enviado\n",mensaje_s.data.nuevo_proc.comando);
                                break;
                        case END_PROCESS_EXECUTION:
                                if(VERBOSE) printf("Proceso terminado %s\n",mensaje_s.data.time.orden);
/*Valido
                                if (sched_policy == CBR) {
                                        actualizaCBR(mensaje_s.data.time.orden,mensaje_s.data.time.gpid_caso,
mensaje_s.data.time.tiempo,mensaje_s.data.time.decision);
                                        if (VERBOSE) printf("Fin actualizacion CBR\n");
                                        }
*/
                                break;
                        case CALC_STATIONS_CHANGE:
                                mensaje_dbm.mtype = READ_ALL_CALC_STATION_MSG;
                                if (sched_policy == NEURONAL_NET)
                                        ajustar_red = 1;
                                msgsnd(dbmq,&mensaje_dbm,1024,0);
                                break;
                        case NEW_DINAMIC_INFO:
                                if ((sched_policy == NEURONAL_NET) && NNET_TRAINING)
                                {
                                        mensaje_dbm.mtype = READ_ALL_CALC_STATION_MSG;
                                        msgsnd(dbmq,&mensaje_dbm,1024,0);
                                }
                                break;
                        case NNET_TRAINING_SWITCH:
                                if (NNET_TRAINING) NNET_TRAINING = 0;
                                else                    NNET_TRAINING = 1;
                                break;
                                }
                }
        }

void terminarAIEngine(int arg) {

        if (sched_policy == NEURONAL_NET)
                terminarRedNeuronal();

        msgctl(schedq,IPC_RMID,NULL);
        fflush(stderr);
        printf("\nClueX AI Engine Finished\n");
        exit(0);
}

void terminar_AI_fault(int arg) {
        fprintf(stderr,"\nAI ENGINE SEGMENTATION FAULT\n");
        if (sched_policy == NEURONAL_NET)
                terminarRedNeuronal();
        msgctl(schedq,IPC_RMID,NULL);
        fflush(stderr);
        exit(-1);
        }


void cambiar_politica(int arg) {

        if (sched_policy == ROUND_ROBIN) {
                sched_policy = QLB;
                printf("\nNueva politica de planificación: QUALIFIED LOAD-BALANCING\n");
                }

        else if (sched_policy == QLB) {
                iniciarRedNeuronal();
                sched_policy = NEURONAL_NET;
                printf("\nNueva politica de planificación: NEURONAL NET\n");
```

```
                }

        else if (sched_policy == NEURONAL_NET) {
                sched_policy = ROUND_ROBIN;
                terminarRedNeuronal();
                printf("\nNueva politica de planificación: ROUND ROBIN\n");
                }



/*      char respuesta;
        FILE *fichero;


        if ((fichero = fopen(SCHEDCONFIGFILE,"r")) == 0) {
          perror("No se pudo acceder al fichero de configuracion QLBconfig.conf\n");
          }
        respuesta = fgets(respuesta, 255, fichero);
        switch(respuesta) {
                case '1':
                        sched_policy = ROUND_ROBIN;
                        printf("\nNueva politica de planificación: ROUND ROBIN\n");
                        break;
                case '2':
                        sched_policy = QLB;
                        printf("\nNueva politica de planificación: QUALIFIED LOAD-BALANCING\n");
                        break;
                default:
                        printf("\nPolitica de planificación no disponible para hot-swapping\n");
                        break;
                }

*/
        }

int mainAIengine(int arg)
{
        int policy=arg;
        sched_policy = arg;

        printf("\nClueX AI Engine Started\n");
        signal(SIGTERM,terminarAIEngine);
        signal(SIGINT,terminarAIEngine);
        signal(SIGSEGV,terminar_AI_fault);
        signal(SIGUSR1,cambiar_politica);
        AIEngine((void *) &policy);
        exit(0);
}
```

MÓDULO IX

BIBLIOGRAFÍA

9.1.- Palabras-clave de referencia

ClueX

Cluster.

Balanceo de carga.

Cálculo distribuido.

Red Neuronal.

CBR.

Planificador

AES.

Agentes.

AIEngine

9.2.- Bibliografía

H. Adeli, S. L. Hung. 1995. Machine learning. Neural networks, Genetic algorithms and Fuzzy systems. 1ª ed. John Wiley & Sons Inc.

K. Ahmed. 2001. Java XML. 1ª ed. Wrox Press.

R.Card. E Dumas. F. Mével. 1997. Programacion Linux 2.0. 1ª ed. Gestión 2000.

H. M. Deitel. P.J. Deitel. 1999. C++. Cómo programar. 2ª ed. Prentice Hall.

B. Eckel. 2000. Thinking in Java. 2ª ed. Prentice Hall

M.K. Johnson. 1998. Linux application development. 1ª ed. Addison-Wesley.

A. López. A. Novo. 1999. Protocolos de Internet. Diseño e implementación en sistemas UNIX. 1ª ed. Ra-Ma.

F.M. Márquez. 1996. UNIX: Programación avanzada. 2ª ed. Ra-Ma

B. Martín del Brío, A. Sanz Molina. 2001. Redes neuronales y sistemas borrosos. 2ª ed. Ra-Ma.

S. J. Metsker. 2002. Design patterns Java workbook. 1ª ed. Addison-Wesley.

T. Mitchell. 1997. Machine learning. 1ª ed. 1997.

E. Rich. K. Knight. 1991. Artificial Intelligence. 2ª ed. McGraw-Hill.

W. Stallings. 2001. Sistemas operativos. 4ª ed. Prentice Hall.

K.Wall. 2001. Programación en Linux. 2ª ed. Prentice Hall.

http://calisto.sip.ucm.es/people/pedro/aa/

http://www.fdi.ucm.es/profesor/belend/ISBC/isbc.htm

http://sopa.dis.ulpgc.es/

http://laurel.datsi.fi.upm.es/~ssoo/FPSO/